

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo No.564

February, 1980

"Threat" Memory

TOWARDS A COMPUTATIONAL THEORY OF SEMANTIC MEMORY

Lucia M.Vaina

Abstract: Research in memory has been a frustrating task not least because of the intimate familiarity with what we are trying to understand, and partly also because the human cognitive system has developed as an interactive whole; it is difficult to isolate its component modules - a necessary prerequisite for their thorough elucidation. Memory cannot be studied in isolation since it is essentially only an adjunct to the proper execution of our ordinary information processing tasks. In order to try to formulate specifically some of the basic requirements of memory we must therefore examine the structure of the processing tasks for which it is used.

This research is oriented toward the design constraints on a memory that is capable of carrying out information processing tasks of the following kind: (1) the assembly of descriptions of objects; (2) the assembly of descriptions of actions (represented in the use of objects, or their function); and (3) certain limited kinds of inference; (4) the assembly of objects in more general categories.

Introduction Memory is one of the most important functions of the human brain, yet its understanding -why and how it does what it does- has so far eluded us. Research in memory has been a frustrating task not least because of the intimate familiarity with what we are trying to understand, and partly also because the human cognitive system has developed as an interactive whole; it is difficult to isolate its component modules - a necessary prerequisite for their thorough elucidation.

Before one embarks on a new research area its general demands and underlying theoretical hypotheses should be stated.

Fundamental demands of research in memory

The first demand to the student of memory, and of cognitive abilities in general, is to attempt to separate those topics that appear capable of explanation by available approaches from those for which no ready explanation even in outline seems available.

The second demand concerns the properties and attributes of the human brain that the researcher must aim to account for. Some of these are:

- 1) the redundancy and self-restorative nature of the brain
mechanisms connected with learning and memory
- 2) the capacity to learn
- 3) the modification and refinement of the information already
stored
- 4) the local character of computations (a local change shouldn't
require large modifications)
- 5) the existence of sensory-specific modules that independently

convey information to the language module.

The third demand concerns to the evaluation of the research done and the choice of those methods that seem to lead to the most relevant results. It has become clear that the most reliable approach to the study of the brain activity is to regard it as a large and complex information processing system. Central to this approach the belief that human cognitive capacity can fruitfully be viewed as some kind of symbolic system. Thus much is to be learned by developing computational theories for aspects of human information processing and comparing the results of these theories and their implementations with human performance on the same tasks. Thus, behavioral phenomena may suggest or constrain possible information processing tasks whose properties can be studied computationally, and thus might lead to the search for previously unrecognized behavioral consequences that they imply.

In an information processing approach, such as the one to which I subscribe, there is a distinction (pointed out by Marr & Poggio [6]) between the various levels at which our information processing device may be understood: at one level, there is the theory of computation, which is what is computed and why, and at the next level is the particular algorithm, or way in which the computation is carried out. The present goal is to elaborate the computational theories of some of cognitive abilities of the human brain. The particular implementation, although eventually important, plays only a secondary role at the moment.

Theoretical hypothesis of research

1) To understand various disabilities resulting from lesions to the brain helps us to understand the normal function of the brain.

2) Data are important for the process of developing the theory; ideas and hypotheses that are at the variance with the data have to be rejected.

3) We should bear in mind that the facts that we deal with are soft and the working domain ill understood, so our intellectual resources would be misplaced if at the present they are spent on the construction of elaborate mathematical structures.

Levels of research into memory

Memory may be studied at several levels. At the most physiological end one would study the neural basis of memory, how the hardware implements the storage process[Examples of theories at this level are the cerebellum [4], [8], the mathematical theory of associative memory devices [7]. Although such research can provide us with illuminating insights into the functioning of the brain, and provides an essential component of an eventual understanding of memory, it is clearly not the whole story. Although studies at this level address the details of how to implement a certain kind of memory in a particular hardware, they unfortunately shed no light on what information should be stored in the memory or how it should be represented there. The underlying reason is that studies at this level contain no analysis of the uses of memory in the broader context of day to

day information processing tasks.

Memory cannot be studied in isolation since it is essentially only an adjunct to the proper execution of our ordinary information processing tasks. In order to try to formulate specifically some of the basic requirements of memory we must therefore examine the structure of the processing tasks for which it is used. A first division in central processing, although a rough one, would be between modality specific and modality unspecific analysis. Examples of modality specific analysis include the tasks of visual analysis, tactile analysis, auditory analysis, etc.. It is clear that these different types of analysis must be taken at least some way before cross-modal interactions of any complexity could be useful, and in fact clinical evidence from neurology suggests that these analyses can proceed a substantial way before their combination. Thus in vision, for example a sophisticated representation of the shape and disposition of a viewed object can be derived by patients whose realization of the shape's use or purpose is severely impaired.

Each of these modality specific analyses poses its own self-contained memory problem, whose primary purpose will be to aid the recovery of a structural description (in the case of vision, of the shape of the viewed object from images of it). One might call such memories modality specific recognition memories (MSRM). Thus in vision, for example, the MSRM which Marr and Nishihara [5] used to organize and store their 3-D models, (see fig.1), is deployed during the construction of a specific, arbitrarily detailed, object-centered description of the shape of the

viewed object. According to current thinking, visual processes preceding this step do not usually involve the deployment of a learned catalog of shapes; they consist almost exclusively of memory-free perceptual processes, like stereopsis and structure from motion, and can usefully be thought as pure perceptual processes.

If this were generally true, one could view the different recognition modules as roughly consisting of two parts: the first, which one might perhaps call pure perception, consists of essentially memory-free analysis of the incoming sensory information; and the second involves the use of memory of stored descriptions during the construction of a representation of the incoming information. Thus in vision, for example, a patient lacking his visual recognition memory but retaining his perceptual apparatus should still be able to perform simple visual tasks, like discriminate two lines at different orientations, or two points at different depths, even though unable to describe the shape of the viewed object. The descriptions supplied by the different MSRM, are potentially complex, since they are capable of representing exhaustively all the information that can be acquired via that particular sense. For example the description from vision of even a fairly simple shape can include 3-D models for aspects of the shape at several different scales, as illustrated in Fig.2

Yet rich as these individual modality specific descriptions can be, we know from our own experience that the comprehension of what we see, touch, hear, involves more than each one, and more even than their

inferential mechanism.

The information in semantic memory is organized and can be modified, and the modifications in semantic memory are caused by interaction with the environment, for example through the Modality-Specific Recognition Memories and probably other Cognitive Memories, and they are produced by various internal processes, for example by inference. Because of this, semantic memory must be organized so as to allow modifications without requiring dramatic changes in its overall structure; that is, modifications for the most part to be local, without entailing global consequences. This requirement could be formulated as a principle of the modularity of the representation. The computational investigation of semantic memory enquires about the nature of the representations used by the cognitive system for storing knowledge about the world, their structure, and the processes by which they are derived, accessed, and preserved.

Theoretical fondation and description of thread memory

The direct motivation for thread memory was a paper by E.K.Warrington, entitled "The selective impairment of semantic memory"[11]. Warrington studied in detail three patients, selected on the basis of a failure to recognize or identify common objects on visual confrontation. This deficit, called by Lissauer "associative agnosia", manifests itself by the patient's being generally capable of describing, copying and representing

the stimulus item, yet having no knowledge of its use or purpose. Warrington argued that this is a deficit of semantic memory, characterized by the degradation of stored information, knowledge of subordinate categories being more vulnerable than knowledge of superordinate categories.

Another set of data comes from clinical studies of anomia. Anomia is a failure of naming on confrontation. As Geschwind [1] pointed out, one must make sure that the failure to name an object is not the result of a failure of perception, or comprehension. Poor naming is a characteristic of aphasics in general and it is manifested most dramatically in anomics. In general when confrontation naming is impaired, the patient, while his speech is fluent, uses many circumlocutions or over-general words, such as "thing", "place", etc.. Anomia is generally present for all stimulus modalities, but the intensity of the disturbance [2] is not necessarily equal for them all. A question that naturally arises here is to what extent is anomia due to a failure in the individual recognition memories and to what extent is it independent of them.

In a collective study by Goodglass et al. [2], it was shown that preservation of letter and number naming were more characteristic of patients with posterior speech zone lesions; these were patients in whom anomia, particularly for object names, was very characteristic. Broca's aphasics, patients with more anterior lesions, are characterized by telegraphic speech and lack of function words, yet they seem to be much better at naming. This observation, and other similar ones, lead to the

question of whether there is a difference between naming errors for different kinds of objects (e.g. singular terms, mass terms, proper nouns, etc.).

In a recent study by Goodglass [3] it was shown that Broca's aphasics have a relatively high proportion of concrete or picturable words. Fluent aphasics, on the other hand apparently preserve a higher proportion of abstract words. Goodglass conjectured that this was because fluent aphasics readily produce the grammatical context leading up to the key nouns in their messages. This pattern would be expected to give them an advantage over Broca's aphasics for the production of nouns that are highly context determined. Thus he supposes that the marked preference of fluent aphasics for abstract over concrete words relies on their facility for producing contexts that favour high frequency abstract words like time, year, week and so forth.

In their paper, "Naming in Aphasia: Interacting effects of form and function"[12], Whitehouse, Caramazza and Zurif asked whether anomia was the same process in all states of brain damage in which it appears. They pointed out that the information-processing chain which generates the labels for a visually perceived object can be disrupted at different stages. They produced evidence that suggests that at least one source of difficulty in naming is a disturbance of the conceptual representation associated with a label. However it is also possible that brain damage spares the conceptual structures underlying lexical organization, but dissociates and disrupts mechanisms responsible for addressing and /or

retrieving information from this structure. Another interesting question is how different categories of aphasics integrate different kinds of information. Broca's aphasics, for example, demonstrated a relatively intact understanding of lexical structures, and have a pretty normal ability to integrate perceptual and functional information. They also seem to be quite good at dealing with the fuzziness of conceptual boundaries. Anomics on the other hand, show evidence of lexical distortion, and manifest an inability to integrate perceptual with functional information. In addition to this, they seem to be insensitive to category boundaries.

The relevance of aphasia to thread memory

The phenomena described in the last section together with my own experience with aphasic patients, have led to the following preliminary formulations of some principles of semantic memory.

- 1) The existence of the different agnosias suggests the presence of several structurally distinct modality-specific representation systems. That is, a particular object is represented internally in several different memories.
- 2) From the work done on visual- recognition memories, it appears that the description of an object is structured rather than unitary, and the units of description are organized hierarchically from the general to the particular.
- 3) There are several ways of representing knowledge about an object: by its modality-specific descriptions, its functions or uses, and the

categories to which it belongs. Each of these modes can be independently impaired. This suggests that semantic memory contains at least three modules, one for each of these modes of representation.

4) Perceptual classifications and semantic classifications are separate, hierarchically organized systems and can be differentially impaired.

5) Most concepts have both superordinate and subordinate categories. But at their most detailed level the representations of two different objects will be different.

6) There is rarely an all-or-none response to an object; on the contrary, some semantic meaning is often preserved, and this is invariably of a general rather than of a specific nature. For example, an agnosic patient in Warrington's experiments would respond faster to "Is a duck an animal?" than to "Is a duck a bird?".

7) From the comparative study of different type of patients (for example anterior vs posterior lesions) we see that the capacity for semantic categorization may be good, the recognition of functionality of objects may be good, and their perceptual description may also be good, yet some types of object may be easier to name than others. Some aphasics do much better with concrete than with abstract objects. This might be due to impaired access mechanisms rather to the degradation of stored information. Thus we can begin to differentiate between processes that access information and processes that store and organize it, and there is some evidence that they can be independently impaired.

8) We have seen that semantic memory as opposed to episodic memory relies

on inferences. So, we can further differentiate the processes that store and organize the information, into processes of inference and processes of storage.

9) From a comparason of posterior with anterior aphasics, it seems that the anterior lesions (Broca's) produce an impairment of the access processes (probably damaging control processes that deal with contextual differences) whereas posterior lesions may impair mechanisms that store and organize the information. Anomic patients can apparently retrieve a superordinate category of the object they are asked to name together with a correct description of its use. They are however apparently often unable to access a more particular representation. (e.g. for a rose, they may get "flower" but not "rose"). It seems then that the damage here is to the inference processes and not necessarily to the storage processes. Wernicke's aphasics on the other hand, seem to have a general impairment of the memory itself.

10) Evidence for the difference between impaired access mechanisms and damage to the memory itself has recently come, for example, from patients at the VAH- Boston. One of them initially had a severe inability to manipulate symbolic expressions, and showed no use of semantic memory. On re-learning the ability to manipulate symbols, some of his use of semantic memory returned. This suggests that his primary damage lay in his access mechanisms.

11) Frequency and familiarity plays an important role in the case of impairment of semantic memory. Frequent or familiar terms are retrieved

correctly more often than less frequent and unfamiliar terms.

Thread memory

The requirements formulated in the previous paragraphs have led us to formulate a new type of semantic memory, called "thread memory". A preliminary version of thread memory was introduced by Vaina & Greenblatt ([10]1979), and the ideas behind it have been evolving over the past year. I will now describe briefly the structure of the memory and some of the processes associated with it.

Structure of the thread memory.

The basic element in the representation of objects in memory is called a thread and it consists of a set of symbols (nodes), ordered according to precise rules. The relation between symbols might have different meanings, each of these meanings being associated with a module of representation. Three modules may be distinguished:

1) a category module, which is a hierarchical organization of the symbols in memory, from the more general category to the more particular category. For example mallard -> thing -> living-thing -> animal -> bird -> duck -> species-of-duck -> mallard

2) a functional module, which contains information about the function, or the uses of the objects. In this module, actions and objects are represented in relation. Thus for example (S LEADS-TO) SEE -> BOY -> RUN.

3) a descriptive module, which contains information about the component

parts of an object, about its appearance. Each element of this module results in the creation of a thread. This module establishes relations between various threads that can be associated with a symbol. For example (BIRD FEATHERS) relates the thread keyed on FEATHERS to the symbol BIRD.

The first element in the thread, called the key, is the stimulus by which the thread is accessed. The threads end in the same symbol as the key. The difference is that when we reach the access symbol at the end of the thread it is loaded by then with all the meaning that is represented on the thread. The symbols in the thread are not unique to the thread, they might appear in several threads. This is a very useful property because it allows a leveled partitioning of information in the memory. So for example, most of the objects from the world could have as their most general representation a symbol like "thing". But that it would not tell us anything, we need to be more specific. The extent to which the memory gets specific can vary, but a semantic memory has to be able in principle to give a unique description to every object in the world.

Objects are represented by a set of threads that give a multiple description of the object and its functionality. The set of all threads associated with an object, or in other words, having the same key, constitute a general thread (Figure 3). There are many symbols in common among the threads in a general thread. Some of them are explicit and, some of them as we will see in the next section can be deduced by the application of the simple thread operation of deduction. A complete

semantic representation of an object is given in all the modules of the semantic memory.

We call the more specific common node of two (or more) threads in the same module a fork-point. The fork-point of all threads in a general thread constitutes the most common knowledge that one has about an object and which is not dependent on its particular occurrence. In each general thread there are different fork-points, depending on the threads that one is looking at. At the fork point one has to decide which representations or descriptions might contain the needed specific information and choose that thread. The fork point of threads that belong to different general threads reveals what the compared representation have in common in a particular module of representation. For example, what is common between a chair and a table in the category module? The usual response is that they are both pieces of furniture. Fork-points obtained in the descriptive module might be misleading for interpretation in another module, like the categorial module for example. Thus a WHALE in the descriptive module is represented as LIVES-IN -WATER and its shape description is like the shape of a fish. From this, one might believe that a whale is a kind of fish. To avoid confusions, simple threads are formed to tell us that a whale is not a fish for example. Or specifying that IS-NOT-A PLANT ANIMAL enables us to distinguish all plants from animals by means of a single thread operation.

Sometimes we need to particularize an object, Thus, dog is a generic term that refers to all dogs in general, Spot is a particular one. Animal is a

generic term too, and Spot is an example of an animal as well as an example of a dog.

The functional module ought for example, to contain things like a knife is an example of "a sharp thing used to cut with". Or when we look for a table, in the absence of it, we could use whatever object that "has a flat surface, big enough to eat on", so a rock could serve this purpose.

Computations within thread memory.

Three main types of computations can be distinguished in thread memory: access processes, inference processes and storage processes. The access processes refer to the interface of the semantic memory with other memories. The inference and storage processes are computations of the semantic memory per se. In talking about each of these processes we see that we can differentiate them further. Thus the access processes are of two types: the access processes themselves and processes that control them. By the general access processes a general thread is brought in the temporary buffer, and then the simple threads are examined one by one (by an operation called "advancing the thread" (Figure 4)). If one had to examine exhaustively every single thread of a general thread the performance would be very slow and uneconomical. The access control processes and some inference processes optimize this. The access control operations can "turn-on" or "turn-off" a group of facts. The context which is "on" at the time of accessing a thread, is always specified, thus in advancing the thread we know that a specific-context, with its restrictions and requirement is

needed. The specific context affects only the extraction or the choice of threads. Once a thread was chosen, and activated in the temporary buffer, the specific context doesn't interfere anymore. We could suppose probably that the specific context is inherited or moved over from the episodic memory.

Another access control mechanism is the general context which allows the threads to be related. The information handled by the general context mechanisms is the information represented in the semantic memory. We can say then that the control access mechanisms control the interface between the information in episodic memory, the information in the semantic memory and the temporarily active buffer.

Inference processes.

In the section about the structure of thread memory the fork-points were discussed as being special nodes on the threads. Finding the fork-point is a special inference process that deals with compare and contrast questions. Another inference process is the process of deduction. For example from the two following threads

ELEPHANT -> ANIMAL -> ELEPHANT

ANIMAL -> LIVING-THING -> ANIMAL

we can deduce the answer to the question: Is an elephant a living thing? The necessary one step deduction is performed by searching each thread whose semantic node is on the elephant thread, for the node living-thing.

After the node ANIMAL is found, a storage operation is performed (the process of assimilation) to store the deduction. An interesting feature of deduction in thread memory is that the depth of deduction is not the distance between two nodes, as in most proposed semantic memories, but the number of jumps from one thread to another(Figure 5a). Thus a very modest search depth can find solutions to non-trivial questions.

There are several ways of optimizing the inference processes. For example, we can define a measure of closeness between symbols in each module of representation called neighbourliness. I think this may be important for problem solving or for the improving the access process that advances the thread. If the result found is acceptable then the neighborhood is reinforced. The neighborliness is not a simple information processing task; it relies on many other operations. For example, crucial in the measurement of neighborliness is the number of threads in a general thread whose fork point is quite deep (we shall see in the storage processes that these threads are "bundled" together Figure 6f). In the functional module the process of neighborliness is used to make generalizations; from particular examples inferences are made to more general categories. This is clearly an optimization process because it actually reduces the enormous number of particular examples of objects that are in the same relation with verbs, to their common fork-point in the category module. For example from

(S LEADS-TO)JOHN CRY

(S LEADS-TO)JANE CRY

(S LEADS-TO)DOG CRY

(S LEADS-TO)BABY CRY

(S LEADS-TO)BOY CRY

(S LEADS-TO)MOTHER CRY

we by applying the neighbourness process we obtain that (S LEADS-TO)ANIMAL CRY and by that we would know for example that we can say ELEPHANT CRY as well as RABBIT CRY.

An interesting observation can be made, namely that the processes of inference are not simple processes, they presuppose processes of storage.(These remarks relate to a hypothesis made in the study of aphasic patients, that in the case of Wernicke aphsics probably all inference processes as well as the storage processes are impaired and yet in the case of anomic aphasics the impairment is of the inference processes. It is supposed in the literature that Wernicke's aphasics who improve, usually recover to become anomic aphasics. This seems very much to support our hypothesis that the processes of inference are based on the processes of storage and the storage processes are first recovered).

Storage processes

The simplest storage process is the process that creates a thread, a new representation in the memory. Usually new information is based on preexisting information, or at least on the descriptive information that is given by the single modality recognition memories, or information from

other cognitive memories. The module in which the new thread is created depends on the way on which the information is acquired. An important class of storage processes is assimilation processes. These assimilation processes chain together intermediate deduction steps so that the entire chain is available as a single deduction step. The assimilation processes build the thread, and make it more complex by storing the result of deductions.(Figure 5b)

There are other types of storage processes such as processes that optimize the thread. First, threads that share a number of common nodes constitute bundles of threads. These bundles are thicker on the upper part of the thread, where the information is less specific, but they divide into smaller and smaller bundles until they break down to single strands. Bundles have a way of recording their thickness. This recording is very important for the accessing of the information. Unless a special context instruction is given, usually the thickest bundle is looked up first. The thickest bundle in a general thread represents the prototypical representation of an object, and it is accessed first. The thickness of bundles is not fixed, for it changes with every incoming thread. Thus the prototype might change too.

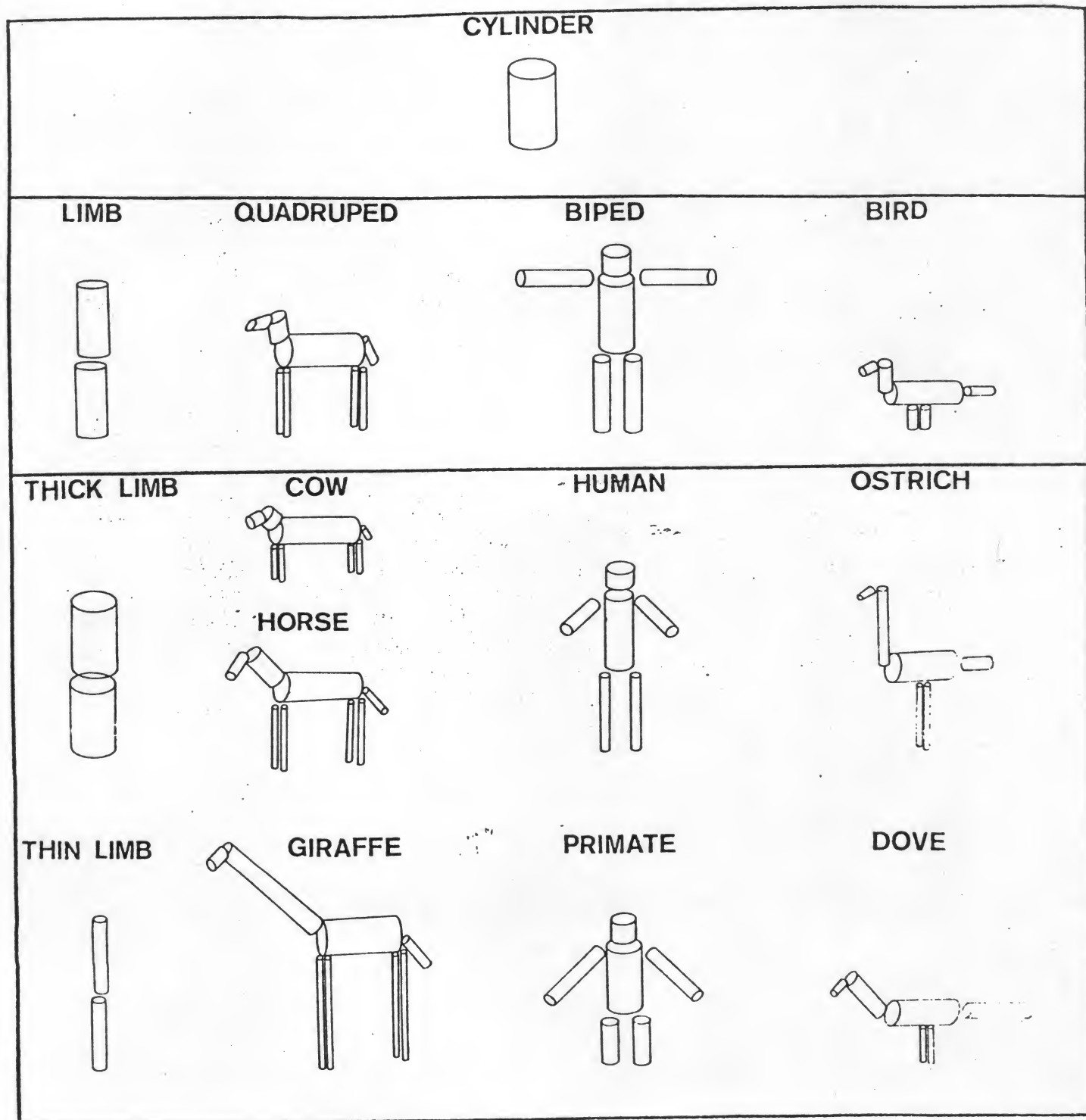
An other optimization process is used when it appears that several threads are sharing a common part. Then it seems useful to make a rearrangement so that all the simple threads share a single pointer to the common part. In other words the uncommon part constitutes a paradigm in the context determined by the common part. The elements in the paradigm behave as they

were the same. We can therefore replace in that context the paradigm with a new symbol, that is more general. For example in the functional module we can have different elements that have the same function, like: one can throw a rock, a ball, a plate, a pencil, a piece of wood, etc. These objects form a paradigm in the context of throw. We can replace them with a common symbol, like: small physical object. This new type of symbol, called micro-concept, will be represented in the descriptive module by the elements that the elements in the paradigm have in common. Thus we see that this new, more abstract concept is derived from particular examples of objects in the real world that can function in certain way (e.g. can be thrown). Micro-concepts are representational units for the object in the real world(Figure 6). They serve for their better understanding and manipulation. They serve also for the interface with other cognitive modules, like for example the modules involving language.

ACKNOWLEDGEMENTS- I am grateful to David Marr, for many stimulating discussions and to Richard Greenblatt, with whom much of the overall framework of thread memory was developed . I thank Dr. Norman Geschwind for carefully reading my manuscript and for his illuminating observations.

Many thanks to the members of the Aphasia Research Center and the speech therapists of the Veterans Administration Hospital Boston, for making my experimental research possible. I especially thank M. Walsh and Dr. Harold Goodglass for their encouraging support.

FIGURE ONE



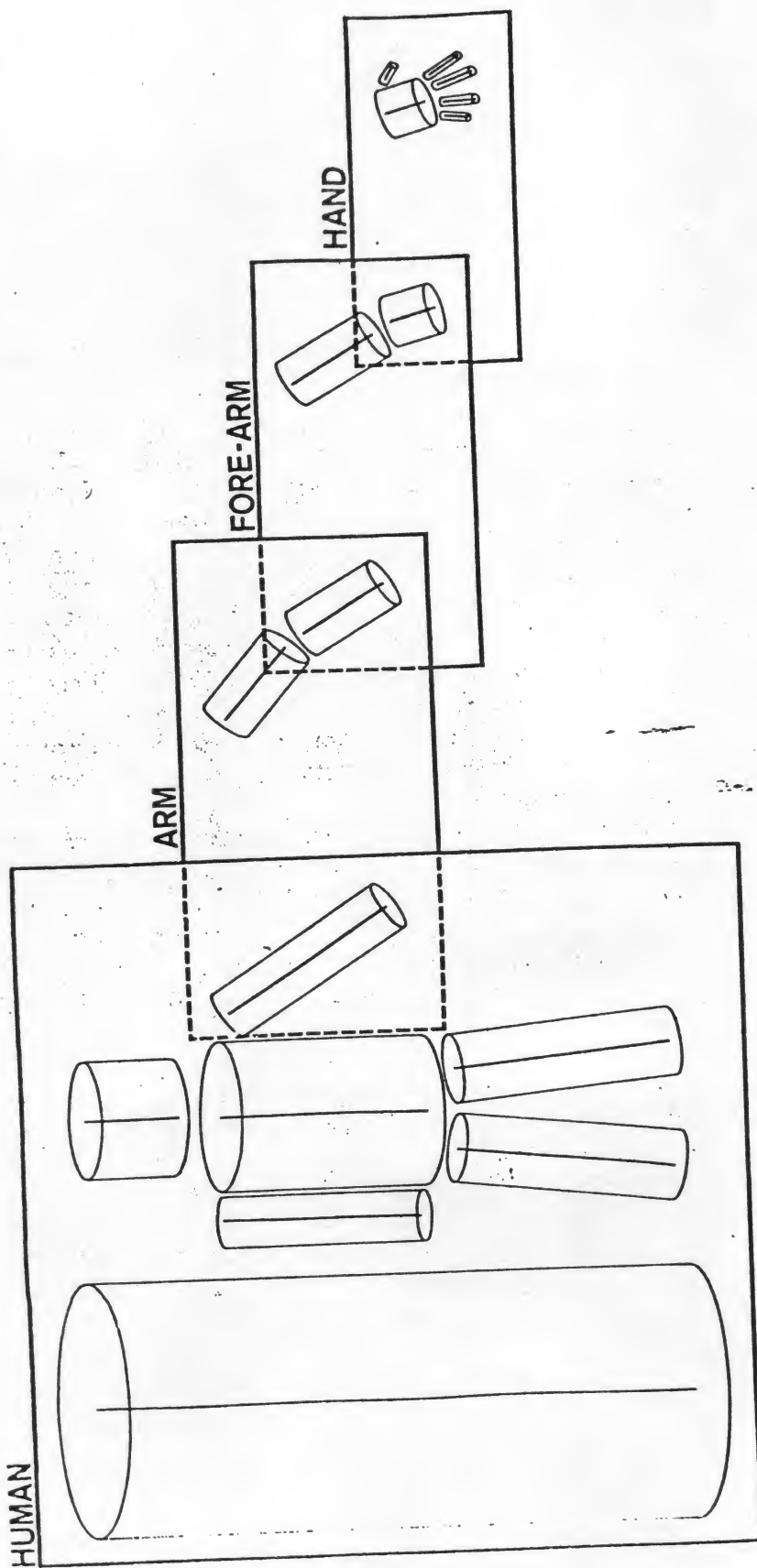
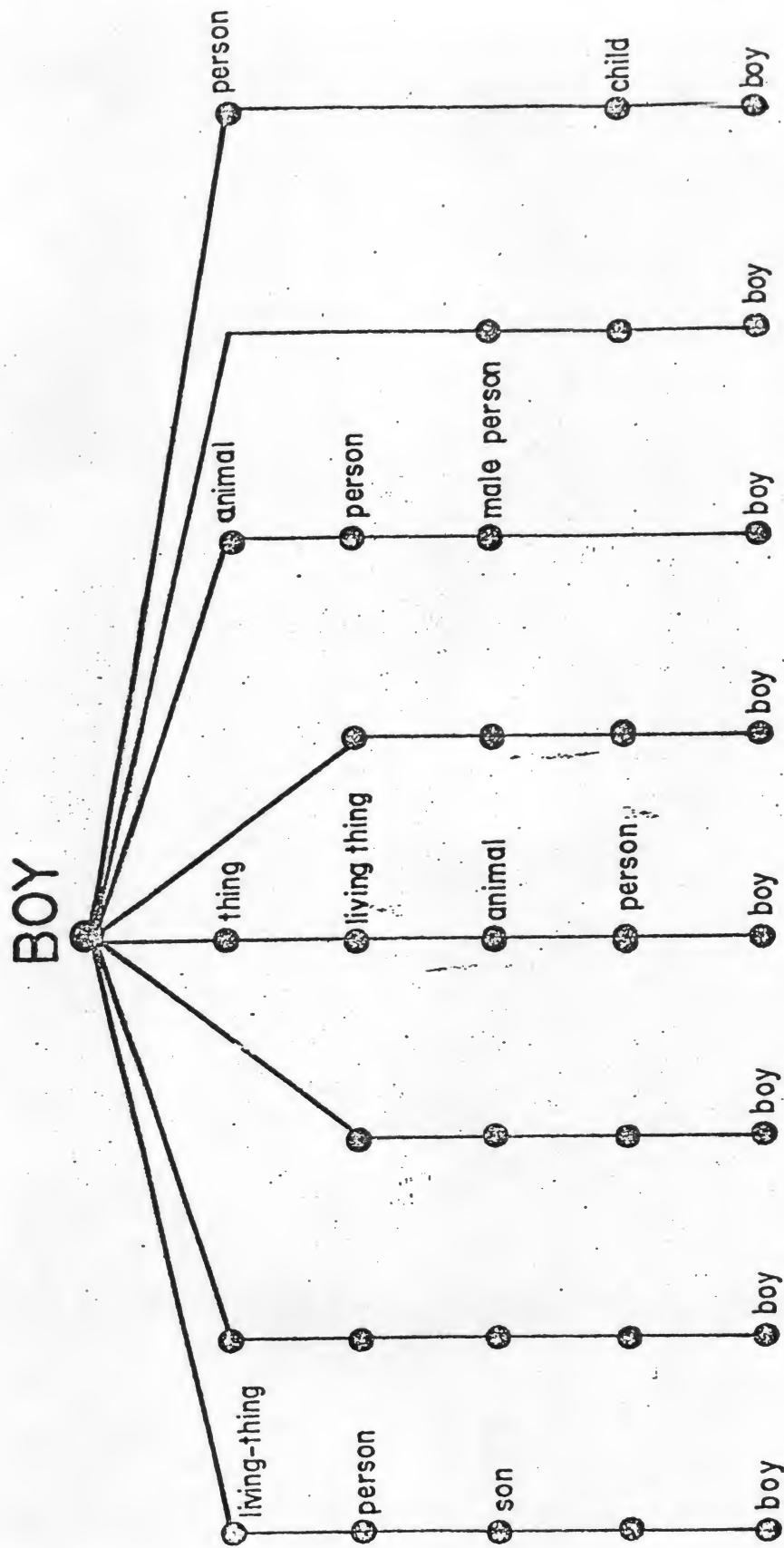


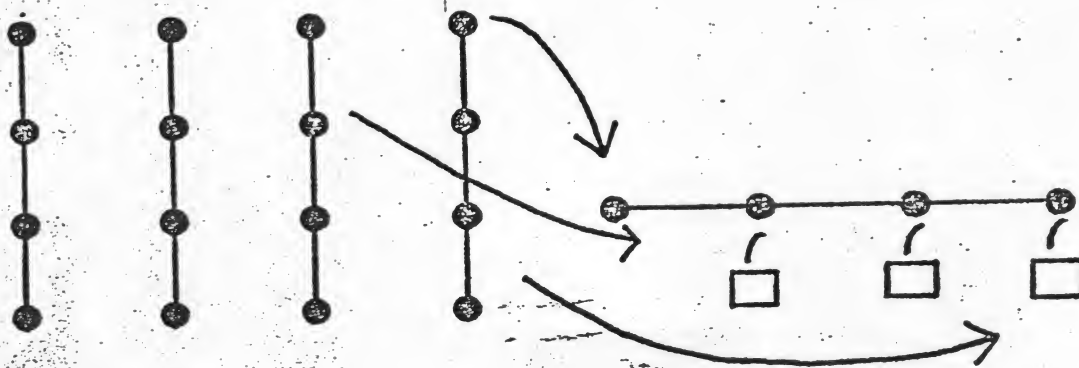
FIGURE TWO

FIGURE THREE



EXAMPLE OF A GENERAL THREAD

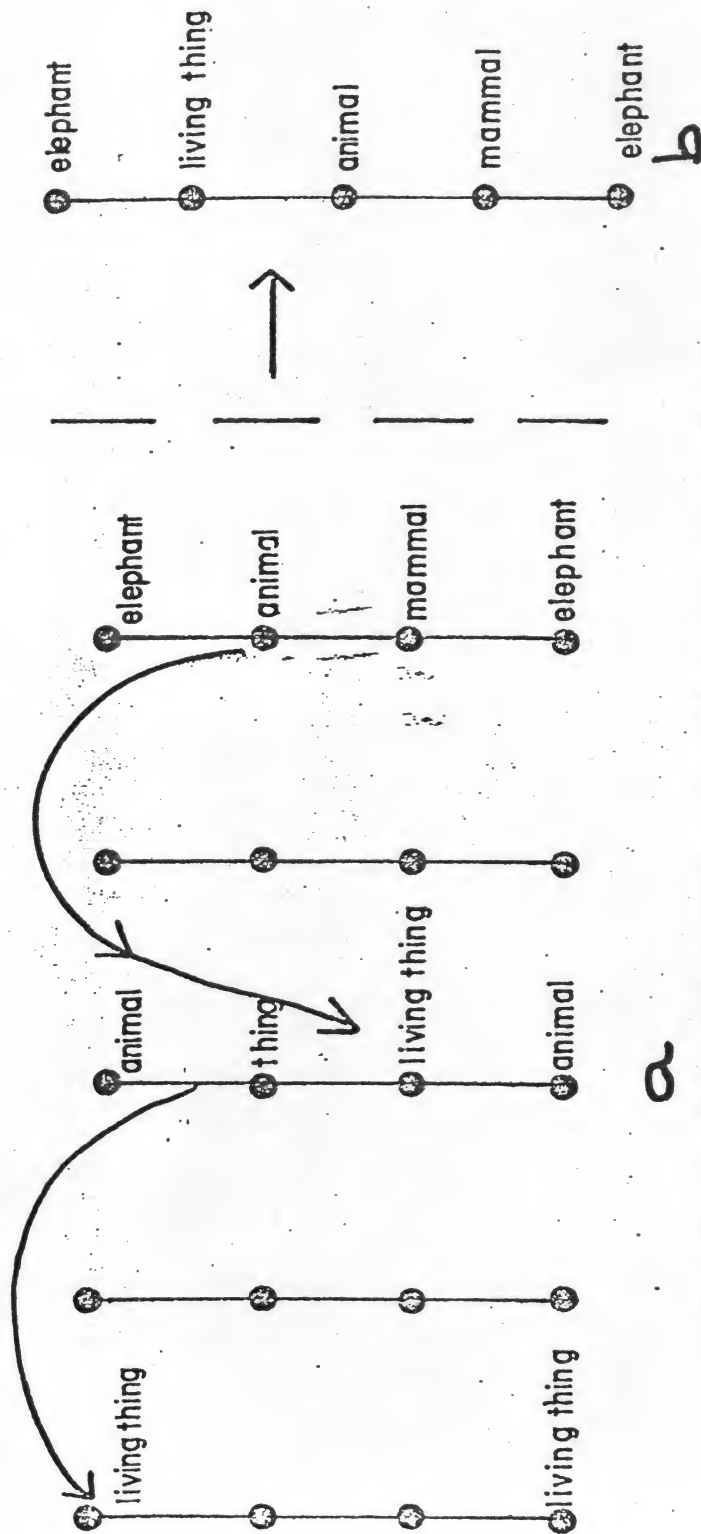
FIGURE FOUR



ACCESS PROCESSES: ADVANCING
THE THREAD

FIGURE FIVE

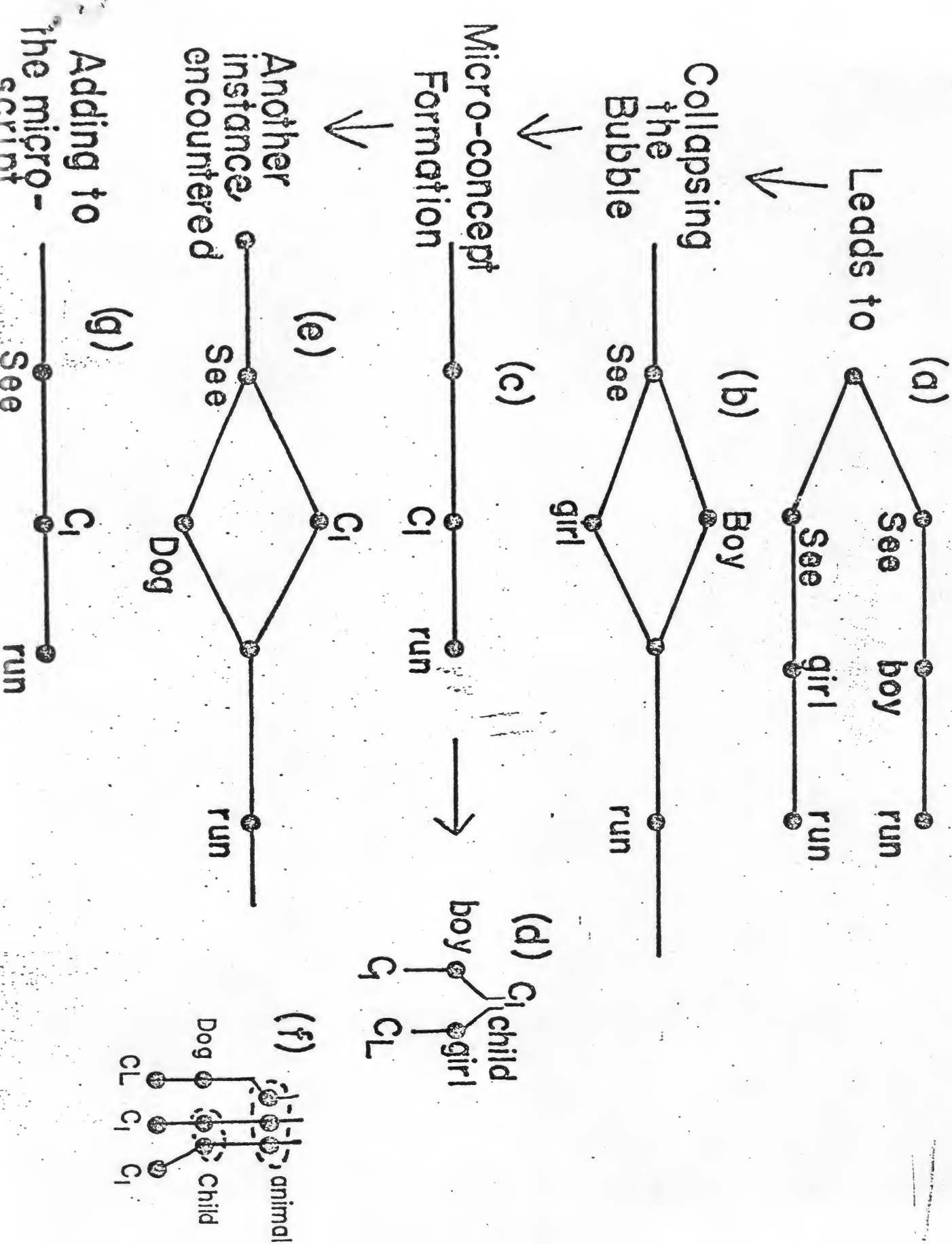
DEDUCTION: Is an elephant a living thing?



Through one step of deduction we obtain the answer, and at the same time the general thread of living-thing (the facts about living-thing).

FUNCTIONAL MODEL

FIGURE SIX



References.

- [1] Geschwind, N. 1967 The varieties of naming errors. *Cortex* 3, 97-112

- [2] Goodglass, H., Hyde, M.R., Blumstein, S. 1969 Frequency, picturability and availability of nouns in aphasia. *Cortex* 5, 104-119

- [3] Goodglass, H. 1978 disorders of naming in aphasia. *Fellow's address, Division 6, American Psychological Association, Toronto.*

- [4] Marr, D. 1969 A theory of cerebellar cortex. *J. Physiol. (Lond.)* 202, 437-470.

- [5] Marr, D. & Nishihara, H. K. 1978 Representation and recognition of the spatial organization of three-dimensional shapes. *Proc. R. Soc. Lond. B* 200, 269-294.

- [6] Marr, D. & Poggio, T. 1977 From understanding computation to understanding neural circuitry. *Neurosciences Res. Prog. Bull.* 15, 470-488.

- [7] Poggio, T. 1975 On optimal nonlinear associative recall. *Biol. Cybernetics* 19, 201-209.

- [8] Szentagothai, J. 1971 Memory functions and the structural organization of the brain. *Biology of Memory* Adam, G. ed. *Symposia Biologica Hungarica* 10, 21-25
- [9] Tulving, E., 1972 Episodic and semantic Memory. *Organization and memory* Tulving, E (ed), Acad. Press New York.
- [10] Vaina, L.M., & Greenblatt, R. The use of thread memory in anomic aphasia and concept learning. 1979 *AI Working Paper* 195
- [11] Warrington, E.K. The selective impairment of semantic memory. 1975 *Quarterly Journal of Experimental Psychology* 27, 635-657
- [12] Whitehouse, P., Caramazza, A., & Zurif, E. 1978 Naming in Aphasia: Interacting effects of form and function. *Brain and Language* 6, 63-74

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A. I. MEMO 410

DECEMBER 1976

VIEWING CONTROL STRUCTURES
as
PATTERNS of PASSING MESSAGES

Carl Hewitt

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0643 and in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0522.

TABLE OF CONTENTS

I	ABSTRACT	1
II	METHODOLOGY	2
II.1	Modeling an Intelligent Person	2
II.2	Modeling a Society of Experts	2
II.3	The Actor Programming Methodology	3
III	THE ACTOR MODEL	4
III.1	Actors	4
III.2	Components of the Actor Model	7
IV	ACTOR CONTROL STRUCTURE	9
IV.1	Introduction to Event Diagrams	9
IV.2	Actor Transmission	11
IV.2.a	Messengers	12
IV.2.b	Envelopes	13
IV.3	Request and Reply	13
IV.4	Recursion	15
IV.4.a	Scripts for a Non-Iterative Factorial	15
IV.4.b	An Event Diagram for factorial Calling Itself Recursively	16
IV.4.c	Snapshot of Storage at Instant when factorial receives [1]	17
IV.4.d	Viewing Recursion as a Pattern of Passing Messages	18
IV.4.e	Characterization of Recursion as a Pattern of Passing Messages	19
IV.5	Envelope Level Scripts	19
IV.5.a	A More Explicit Script for the Non-Iterative Factorial	20
IV.6	Iteration	22
IV.6.a	A Script for an Iterative Implementation of Factorial	22
IV.6.b	An Event Diagram for Iterative Factorial	23
IV.6.c	A More Explicit Script for Iterative Factorial	24
IV.6.d	Meaning of "Recursion"	25
IV.7	Comparison of Recursion and Iteration	26

V	EFFICIENCY and INTELLIGIBILITY	27
V.1	Modular Distribution of Knowledge	27
V.2	Non-hairy Control Structure	27
V.3	Gaining Efficiency thru Progressive Refinement	28
V.4	Generators	31
	V.4.a A High-Level Implementation	33
	V.4.b An Incremental Implementation	33
VI	The LAMBDA CALCULUS of CHURCH	35
VII	FUTURE WORK	37
VII.1	Applications	37
	VII.1.a Incremental Perpetual Development	38
VII.2	The Actor Problem-Solving Metaphor	39
VIII	ACKNOWLEDGEMENTS	40
IX	BIBLIOGRAPHY	42
X	APPENDIX: Introduction to PLASMA	46
X.1	Sequences and Collections	46
X.2	Transmitters	46
X.3	Pattern Matching	48
X.4	Receivers	48
X.5	Conditionals	49
X.6	Definitions	50
X.7	Unpack	52
X.8	Use of Sequences	53
X.9	Delay	54
X.10	Packagers	55

SECTION I --- ABSTRACT

The purpose of this paper is to discuss some organizational aspects of programs using the actor model of computation. In this paper we present an approach to modelling intelligence in terms of a society of communicating knowledge-based problem-solving experts. In turn each of the experts can be viewed as a society that can be further decomposed in the same way until the primitive actors of the system are reached. We are investigating the nature of the communication mechanisms needed for effective problem-solving by a society of experts and the conventions of discourse that make this possible. In this way we hope eventually to develop a framework adequate for the discussion of the central issues of problem-solving involving parallel versus serial processing and centralization versus decentralization of control and information storage.

This paper demonstrates how actor message passing can be used to understand control structures as patterns of passing messages in serial processing. This paper is a pre-requisite for successors which treat issues of parallelism and communication within the framework established here. The ability to analyze or synthesize any kind of control structure as a pattern of passing messages among the members of a society provides an important tool for understanding control structures. Ultimately, we hope to be able to characterize various control structures in common use by societies in terms of patterns of passing messages. This paper makes a small step in this direction by showing how to characterize familiar control structures such as iteration and recursion in these terms.

SECTION II --- METHODOLOGY

II.1 --- Modeling an Intelligent Person

Newell [1962] characterized what has become the conventional metaphor for computer problem solving as follows: *"The problem solver should be a single personality, wandering over a goal net much as an explorer wanders over the countryside, having a single context and taking it with him wherever he goes."* Working within this paradigm, authors of problem solving programs have often relied on introspection as to methods that they would personally use to accomplish the task. Excellent scientific work has been done working within this metaphor. Some of the work has taken the form of writing a program to perform a task which requires a high degree of problem-solving ability in a human. Other work has attempted to model how an individual human actually performs a simple task at an information processing level.

Research in any scientific field is carried out within the framework of underlying theories. A large portion of the research that has been done in the field of Artificial Intelligence has taken the modeling of an artificial human as its implicit goal. An early form of this modelling paradigm was the goal of constructing devices which would pass the "Turing Test". By this test a device is intelligent if it cannot be distinguished from a human by interaction through a teletype. However, the "Turing Test" view of the goal of artificial intelligence has been abused in recent years. Transcripts that appear to be interactions with programs have been published that give a very misleading impression of the real capabilities of the process that produced the transcripts.

II.2 --- Modeling a Society of Experts

Reciprocal communication of a cooperative nature is the essential intuitive criterion of a society.

Edward O. Wilson in SOCIOBIOLOGY

We are investigating the problem solving model of a society of experts to supplement the model of a single very intelligent human. We submit that this change in focus has several beneficial results. It provides a better basis for naturally introducing parallelism into problem-solving since protocols of individual people do not seem to exhibit much parallelism. The change in focus helps to make mechanisms for the communication of knowledge more explicit. Psychologists have found it extremely difficult to discover the communications that occur in the mind of an individual expert during problem solving. Also the justifications for statements becomes more explicit since one expert will often demand explicit justifications for the statements of another expert. It helps make the goal structures of programs more explicit since experts can demand to know why they are being asked to work on a particular task and how this task fits in with other tasks that are being pursued. Furthermore the change should foster better specifications for tasks to be achieved so that appropriate experts can be selected or synthesized.

In these ways we hope to develop the communication mechanisms that are necessary to achieve cooperation between expert modules for various micro-worlds in order to perform larger tasks which call for the expertise of more than one micro-world. Our work is attempting to build on the analysis that has been done by philosophers of science in recent years on the structure of the processes used by scientific societies. In particular the work of Kuhn and Popper and their followers provides us with a large stock of problem-solving ideas. The long term goal is to construct systems whose behavior approximates the behavior of scientific societies. That is, the ultimate aim is to build systems which model the way scientists construct, communicate, test, and modify theories.

11.3 -- The Actor Programming Methodology

We are developing methods to specify the behavior of actors (objects) in terms that are natural to the semantics of the causal and incidental relationships¹ among the objects. That is, we are attempting to develop a transparent medium for constructing models in which the control structure emerges as a pattern of passing messages among the objects being modeled.

Towards that end, we are developing a programming methodology consisting of the following activities:

Deciding on the natural kinds of actors (objects) to have in the system to be constructed.

Deciding for each kind of actor what kind of messages it should receive.

Deciding for each kind of actor what it should do when it receives each kind of message.

Making the above decisions should constitute the design of an implementation. Thus the data structures and control structures of the implementation should be determined by these decisions instead of being determined by the limitations of the programming language being used. This is not to say that the resulting implementation should be unstructured. Rather the structure of the implementation should develop naturally from the structure of the system being modeled working within the conventions of discourse among actors.

Actors are a local model of computation. There is no such thing as "action at a distance" nor is there any "global state" of all actors in the universe. Actors interact on a purely local way by sending messages to one another.

1: Causal relationships are determined by physical causation in activating computational events whereas incidental relationships are determined by the local order of arrival of messages at their destinations.

SECTION III -- THE ACTOR MODEL

III.1 -- Actors

The basic construct of our computation model is the ACTOR. The BEHAVIOR of each actor is DEFINED by the relationships among the events which are caused by the actor.

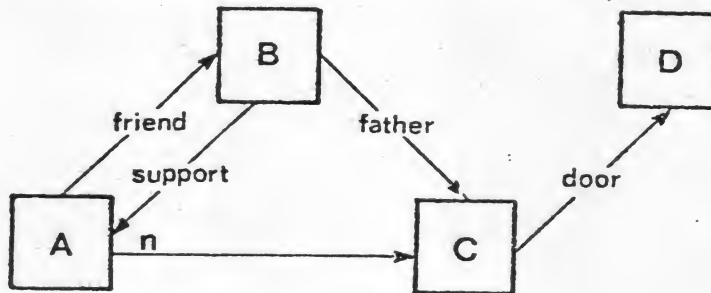
At a more superficial and imprecise level, each actor may be thought of as having two aspects which together realize the behavior which it manifests:

the ACTION it should take when it is sent a message

its ACQUAINTANCES which is the finite collection of actors that it directly KNOWS ABOUT.

We first discuss actors in terms of their physical arrangement because it makes the discussion more concrete and familiar to most readers. Gradually the emphasis will change to a discussion of the behaviors realized by actors.

Diagrammatically we will represent a situation in which an actor A knows about an actor B by drawing a directed arc (which may be labeled for the convenience of the reader) from A to B.



A directly knows about B as "friend"
B directly knows about A as "support"
A directly knows about C as "n"
B directly knows about C as "father"
C directly knows about D as "door"

Diagram of the acquaintances of actors A, B, C, and D

The notation (acquaintances x) will be used to denote the immediate acquaintances of an actor x. For example

(acquaintances A) = {C B}
(acquaintances B) = {A C}
(acquaintances C) = {D}
(acquaintances D) = {}

Note that the KNOWS ABOUT relationship is asymmetric; i.e. it is possible for an actor A to know about another actor C without C also knowing about A. Should it happen that A and B know about each other then we will say that they are MUTUAL ACQUAINTANCES.

The acquaintances of an actor are an abstraction of its physical representation. Consider for example a list L with first element X and rest Y

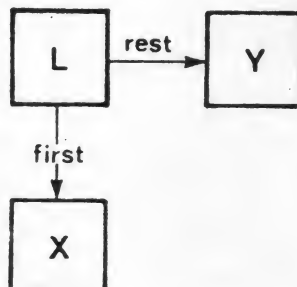
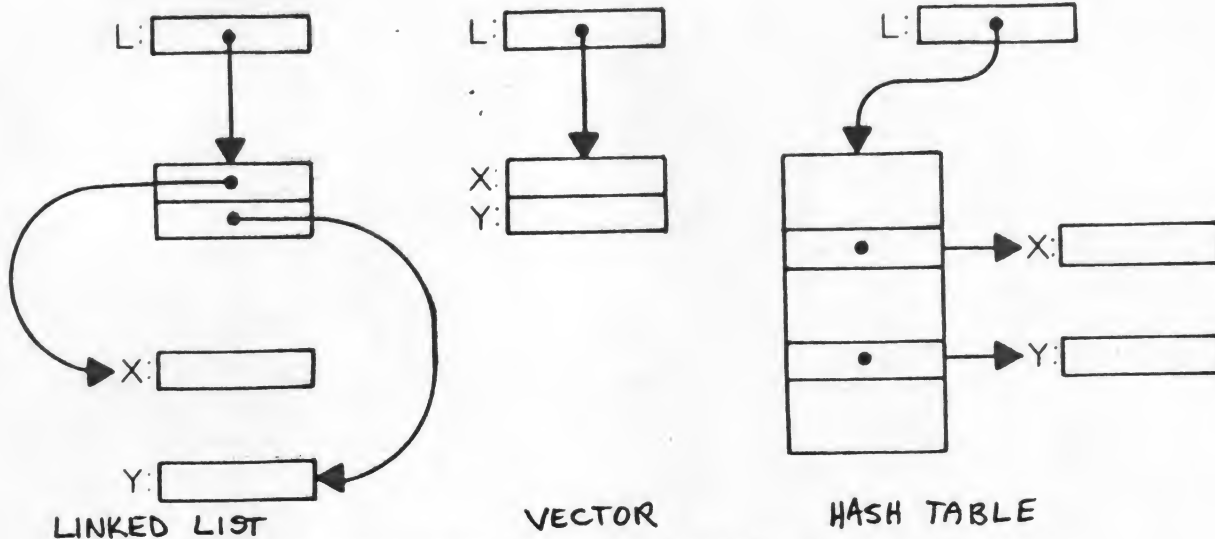


Diagram showing L directly knows about X as "first" and Y as "rest"

The actual physical representation of L could be in terms of a linked list, a vector of storage, or even a hash table:

Diagram showing alternative physical realizations of L

Actors are straightforward to implement on conventional machines. We will mention a couple of ways to do this in order to add concreteness to our discussion. Practical implementations are particularly easy to construct using list-processing languages and micro-processors. Our implementation of actors in LISP uses one cons pair for every actor. One component of the pair is a LISP procedure which provides an entry point into the machine code necessary to implement the behavior of the actor when it is sent a message. The other component of the pair is an ordered list of the acquaintances of the actor. A similar representation could be used on a micro-processor (such as the CONS micro-processor of Knight et. al.). A reference to an actor on a micro-processor would in general require one word of memory which consisted of two sub-fields. One field would be used as an index into the micro-code and the other field would be used to point to a vector of the acquaintances of the actor.

The reader should keep in mind that within the actor model of computation there is no way to decompose an actor into its parts. An actor is defined by its behavior; not by its physical representation!

III.2 --- Components of the Actor Model

The actor message-passing model is being developed as four tightly related and mutually supportive components:

1: A method for the rigorous specification of behaviors from various perspectives. An important degree of flexibility available in actor semantics involves the ability to carefully control the articulation of detail to be included in specifications. That is, the constraints on the behavior of a system of actors can be specified in as much or as little detail as is germane. Too much detail is distracting and impractical. Too little detail fails to specify important aspects of the desired behavior. The wrong kind of detail deflects attention down fruitless paths. Often the specifications need to be very highly articulated for some crucial aspects of the desired behavior and less so for other aspects. We are developing a methodology through which the desired behavior of a system can be specified by axioms which characterize the relationships among the events which must constitute the behavior of the system. At the highest level these axioms are specifications of what is to be done rather than how. As more detailed constraints of the allowable events are gradually imposed, the possible behaviors which will realize these constraints become more restricted until one is uniquely determined. Conversely, in order to demonstrate that a set of specifications is satisfied by a particular actor, one examines the behaviors of the component actors and demonstrates that the connection of these behaviors realizes the behavior that is required.

2: A system (called **PLASMA** for **PLANNER-like System Modeled on Actors**) implemented in terms of actor message passing that is convenient for the interactive construction of scenarios, scripts, and justifications. A **SCRIPT** is a PLASMA program which can be used to specify the action that an actor will take when it receives a message. In our research we have attempted to investigate semantic instead of syntactic issues. We have designed PLASMA to be a transparent medium for expressing the underlying semantics of actor message-passing. For example the semantics of the "knows-about" relationship for actors dictates that PLASMA must use a particular syntactic rule (lexical binding) for the referents of identifiers. The semantic model specifies that acquaintances of an actor must be specified when the actor is created. PLASMA satisfies this semantic constraint by using the values of the identifiers at the time of creation for the free identifiers in the script of a newly created actor since these are the only actors available to be used as acquaintances.

3: A mathematical theory of computation which can represent any kind of discrete behavior that can be physically realized. Our goal is to have a robust theory whose theorems are not sensitive to arbitrary conventions and definitions. A

theory which will be widely applicable as a mathematical tool is needed for formalizing and investigating properties of procedures. Currently our theory takes the form of a set of laws that any physically realizable actor system must satisfy together with a set of axioms that characterize the behavior of a powerful modular set of physically realizable actors (the primitives of PLASMA) which embody conventions for discourse among actors.

4: The Event Diagrams presented in this paper are a further development of a graphical notation used by Richard Steiger in his masters thesis for displaying relationships among the events of an actor computation. In this paper we use them to show the causal and knowledge relationships that characterize simple control structures such as iteration and recursion as patterns of passing messages. Given an outline of important hypothesized events and causal relations among the events of a particular computation (i.e. a SCENARIO of the intended behavior of the system), event diagrams aid in abstracting scripts of modules that are capable of realizing this behavior. For example we plan to explore the abstraction of the scripts of actors for simple procedures for data structures from scenarios of their intended use. Conversely, they aid in the analysis of an existing system by graphically displaying the relationships among the events occurring in the system for particular cases of behavior. Using the displays available on our time-sharing system, we would like to automate the construction and analysis of event diagrams that have been done by hand in this paper. We would like to investigate the construction of an "eclectic magnifying glass" which provides flexible ways to specify which events and relationships in the history of a computation are to be displayed.

This paper introduces and describes the relationship between Event Diagrams and PLASMA for simple computations that do not involve side-effects. Issues of parallelism, inter-process communication, and synchronization will be treated in subsequent papers building on the foundation provided by this paper. For a mathematical treatment of the actor model of computation see (Greif and Hewitt: SIGACT-SIGPLAN 1975) and (Greif: dissertation 1975). Issues of behavioral specifications are treated in (Greif: dissertation 1975), (Hewitt and Smith: Towards a Programming Apprentice 1975), (Yonezawa: Symbolic Evaluation as an Aid to Program Construction).

SECTION IV --- ACTOR CONTROL STRUCTURE**IV.1 --- Introduction to Event Diagrams**

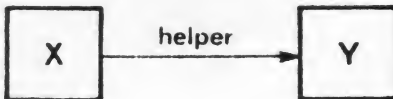
From a strictly input-output point of view there is no difference between iterative and non-iterative implementations of a module. In order to rigorously analyze control structures it is necessary to have a model of computation that is capable of displaying the internal structure of computations.

We shall use event diagrams to display the internal structure of computations. Such diagrams can be used to display many of the significant internal structural relations in a computation. A legend for the notation used in these diagrams is given on the next page.

Legend for Event Diagrams



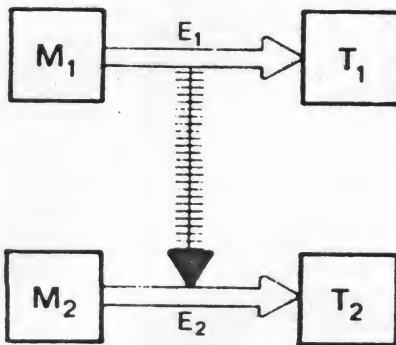
the box represents the actor A



x knows about y as "helper"



the double line represents the **EVENT** which consists of sending the messenger M to the target T



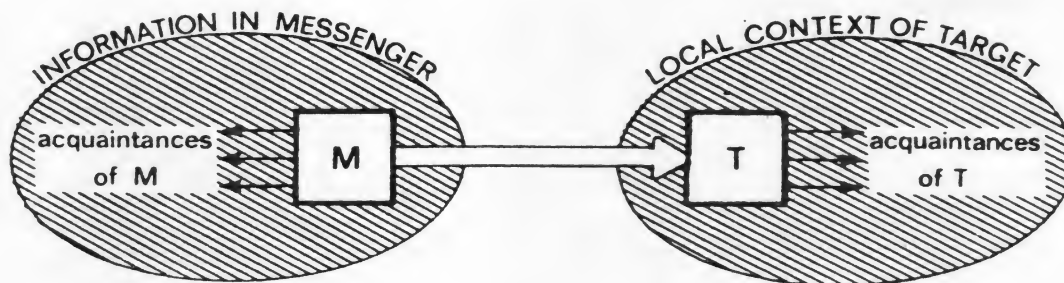
the "railroad tracks" are used to indicate that the occurrence of event E_1 results in the occurrence of the event E_2 and thus E_1 must precede E_2 in time. The event E_1 has messenger M_1 and target T_1 whereas the event E_2 has messenger M_2 and target T_2 .

IV.2 --- Actor Transmission

Actors make use of one universal communication mechanism called **ACTOR TRANSMISSION** which consists of sending one actor (called the **MESSENGER** of the transmission) to another actor (called the **TARGET** of the transmission). Each actor transmission defines an **EVENT** in which the **MESSENGER** arrives at the **TARGET**. The target and messenger are the immediate **PARTICIPANTS** in the event. I.E. if E is an event with messenger actor M and target actor T then

$$(\text{participants } E) = \{M T\}$$

Actor transmission enables the knowledge in the local context of the target actor T to be integrated with the information of the messenger actor M since the acquaintances of both the messenger and target are available for use when the messenger arrives at the target. Furthermore this constitutes the only information available at the instant of computation defined by the event!!!



Event recording the transmission of Messenger M to T

Actor transmission is used to provide the necessary communication between actors to accomplish the following kinds of actions:

- calling a procedure
- obtaining an element from a data structure
- invoking a co-routine
- modifying a data-structure
- returning a value
- synchronization of communicating parallel processes

The actor transmission communication mechanism enforces the modularity and protection of actor systems. It provides the basis for constructing actor systems with explicit modular interfaces such that user of a module (actor) can only depend of the behavior of the actor. The hardware enforces the constraint that the user of a module cannot depend on its current physical representation.

IV.2.a --- Messengers

In order to have a useful model of a message-passing system, the problem of infinite regress must be explicitly addressed. The actor message passing model provides for primitive actors to deal with this problem. When a primitive actor receives a request, it is unnecessary for the primitive to send any further messages in order to properly respond to the request. In particular this means that a primitive actor must be able to obtain some of the acquaintances of a messenger which it receives without having to send any messages. Packagers (see appendix) provide the primitive mechanism needed in PLASMA for transmitting messengers between actors.

Once an actor, m, (serving as messenger) is transmitted to another actor (serving as the target), t, the computation proceeds by following the script of t using information from m. For this to be of any use as a model of communication, it must be that m obeys some fairly standard conventions. These provide the basis for meaningful discourse between actors. We will adopt the convention that all of the messengers constructed by the PLASMA system are packagers² of the following form:

(messenger: (agent: a) (envelope: e) (banker: b))

where a is an actor representing the agent responsible for the computation, e is the envelope of the transmission, and b is the banker funding the computation. The explanation of bankers and agents is outside the scope of this paper so we shall say no more about them.

2: Readers who are unfamiliar with the packagers of PLASMA may wish to consult the appendix.

IV.2.b --- Envelopes

In many cases the envelope of a messenger will simply contain a message. A response to a request is either a **REPLY** envelope with a reply message to the request packaged as

(reply: the-message)

or a **COMPLAIN** envelope with a complaint message packaged as

(complain: the-message)

which explains why the request could not be honored.

Often the envelope of a messenger is a **REQUEST** which in addition to a request message contains an actor c to which a reply to the request should be sent. Such an envelope is packaged as follows:

(request: the-message (reply-to: c))

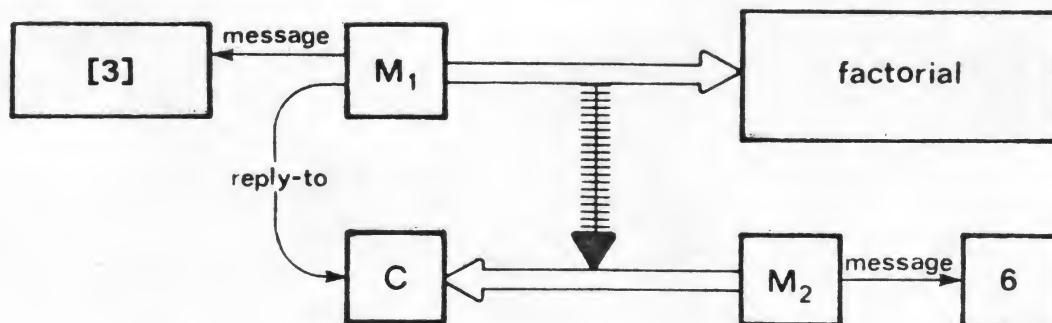
The **ACTOR** c is closely related to the continuation **FUNCTIONS** used by Morris, Wadsworth, Reynolds, and Strachey.

An ordinary functional call to a function f with arguments $arg_1, \dots$, through arg_k is implemented in **PLASMA** by passing to f a request envelope with a message consisting of the tuple $[arg_1, \dots, arg_k]$ of arguments and a continuation actor to which the value of f should be sent.

IV.3 --- Request and Reply

Perhaps the simplest control structure is the ordinary request and reply pattern of activity that is implemented in most programming languages as a procedure call and return. None of the internal structure of the actor being invoked is shown. Instead the description articulates only the input-output behavior of the actor.

Consider the example of a request being sent to an actor *factorial* to compute its value for the argument tuple [3] and send the answer to the actor C. The diagram shows the two events consisting of the above REQUEST (i.e. *factorial* is sent a messenger M_1 with message [3] and continuation C) and the REPLY in which C is sent a newly created messenger M_2 with message 6:



An Event Diagram for the Computation of (*factorial* 3)

The above event diagram treats *factorial* as a "black box" with none of the internal events shown. Notice that the computational process follows the "railroad" tracks from the first event to the second event. We will now proceed to examine the computation more closely. This is an application of the idea of using an eclectic magnifying glass to articulate the description of a behavior in greater detail. What is seen depends on how *factorial* is implemented as well as the focus of the magnifying glass. When we look into the implementation of *factorial*, we will see a number of events that occur between the two which are diagrammed above.

Note that the value 6 which is constructed by the actor *factorial* is not an acquaintance of *factorial*. Instead it is the "reply" acquaintance of the messenger M_2 which is sent to the continuation C.

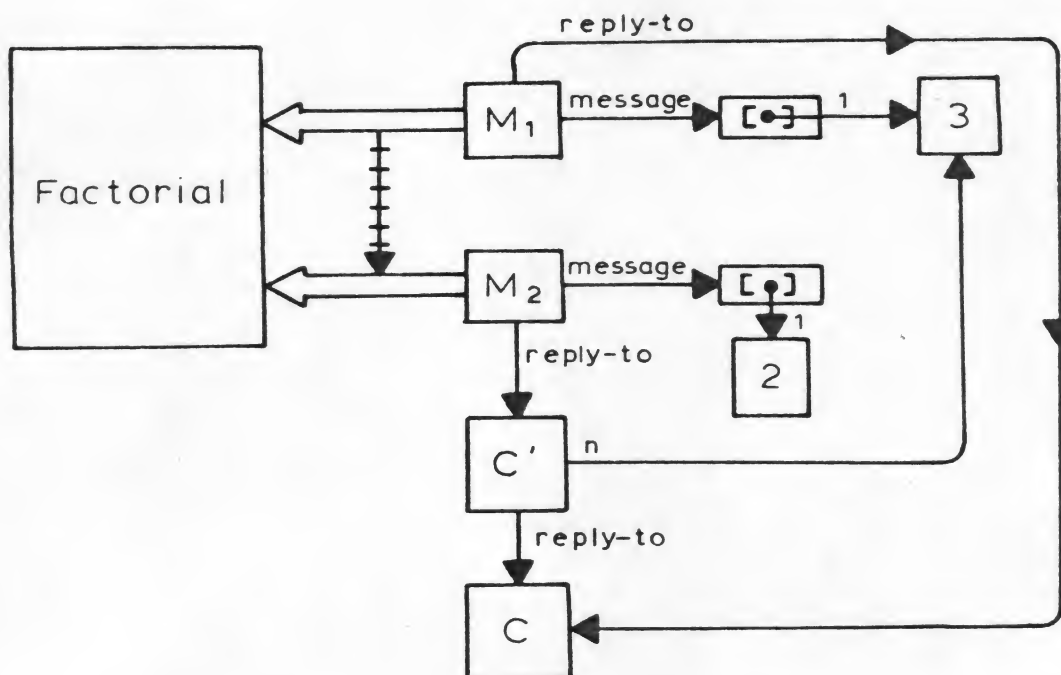
IV.4 -- RecursionIV.4.a --- Scripts for a Non-Iterative Factorial

Suppose we have a non-iterative implementation of factorial. A script written in PLASMA for such an implementation is given below. Readers who are unfamiliar with the notation can consult the appendix which provides an informal introduction to PLASMA.

(factorial ≡	<i>;factorial is defined to be</i>
(≡> [=n]	<i>;receive a message with one element which will be called n</i>
(rules n	<i>;the rules for n are</i>
(≡> 1	<i>;if it is 1</i>
1)	<i>;then return 1</i>
(≡> (> 1)	<i>;else if it is greater than 1 then</i>
(n * (factorial (n - 1))))))	<i>;return n times factorial of n minus 1</i>

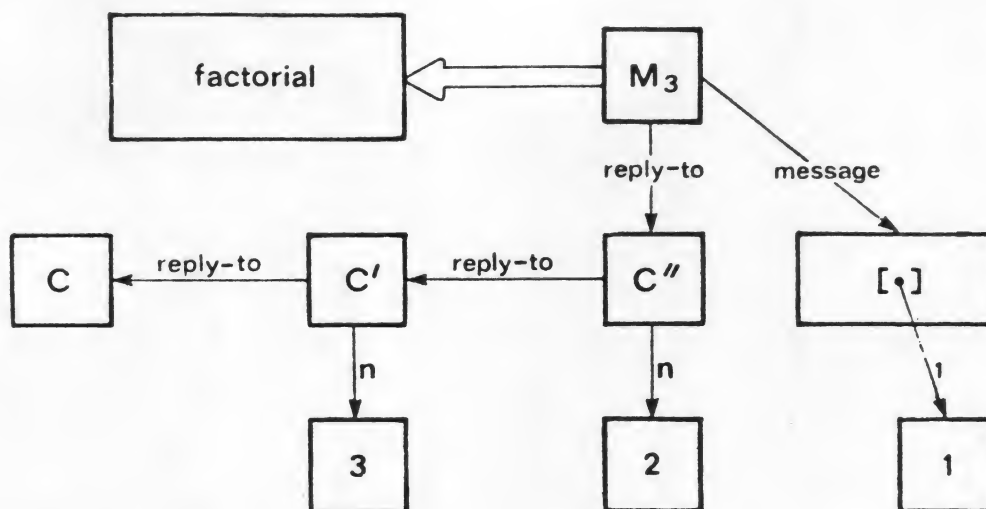
IV.4.b --- An Event Diagram for factorial Calling Itself Recursively

We are interested in looking more deeply into the control structure of recursive procedures. To this end we take the above non-iterative implementation of factorial as a concrete example to be studied. When factorial receives the message [3] it is not able to reply immediately since it does not directly know what (factorial 3) is. Below is an event diagram of the computation that results from sending factorial a messenger M_1 with message [3] and continuation C up to the point of the first recursive call in which factorial is sent a newly created messenger M_2 with message [2] and continuation C' where C' is a newly created actor that knows about n and C . The script of C' is such that whenever it is sent a message y , it sends C the message $(3 * y)$.



IV.4.c --- Snapshot of Storage at Instant when factorial receives [1]

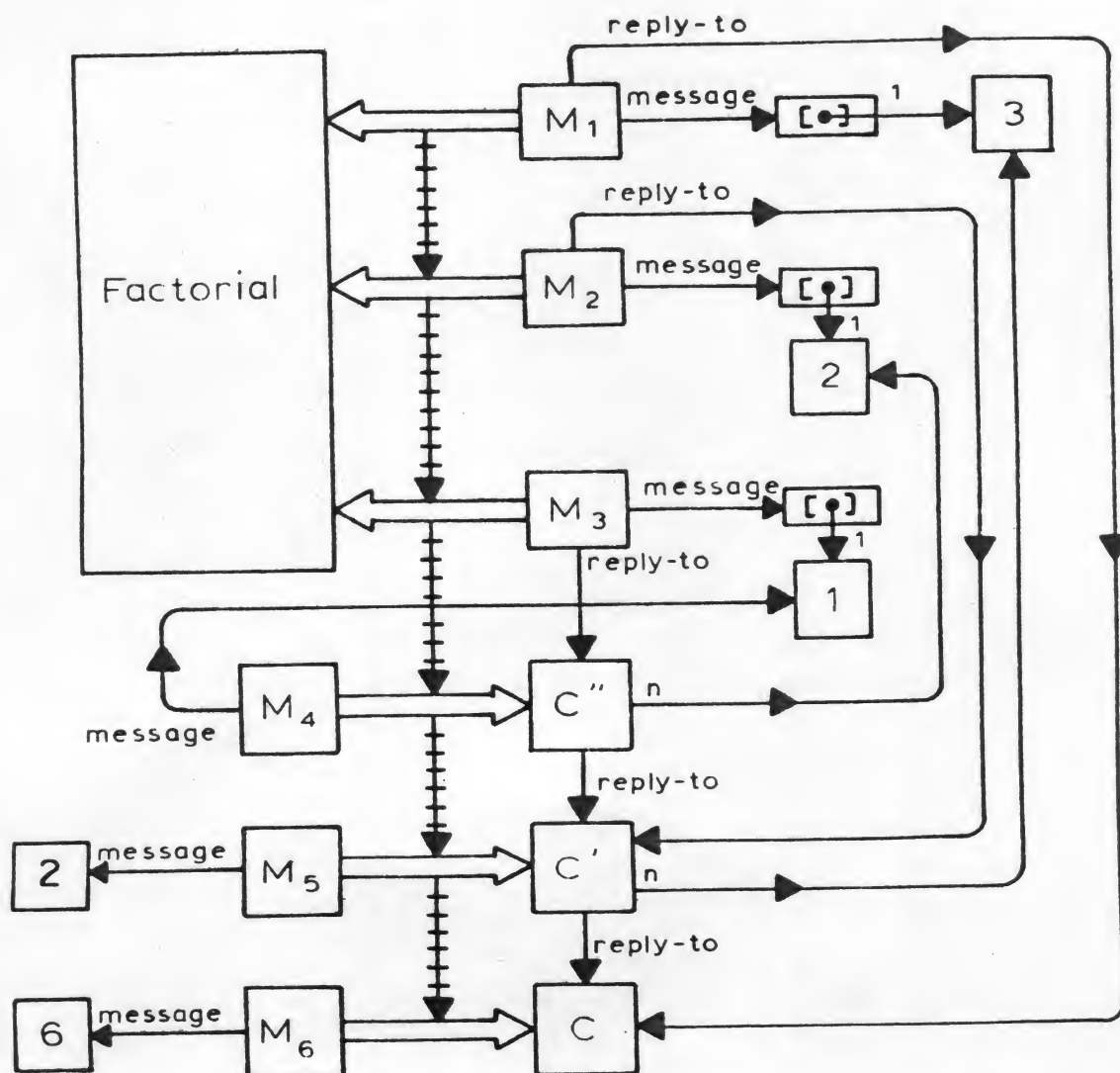
Below we present a snapshot of the storage at the instant factorial receives the message [1]. The rule for computing the amount of storage being used at the instant of any particular event is very simple: Merely count all the actors that are in the transitive closure of the acquaintances of the participants involved in the event. Recall that the participants of an event are the actors immediately involved (i.e. the target and messenger).



IV.4.d --- Viewing Recursion as a Pattern of Passing Messages

The above event diagram exhibits the characteristic structure of a recursive computation. This pattern is familiar to users of ALGOL, LISP 1.6, and PL-1 and other programming languages that make use of a pushdown stack to implement recursion. In such languages the amount of stack used by the implementation grows monotonically until factorial is called with the argument 1 and then monotonically decreases as the stack is popped.

Below we give an event diagram that displays the pattern of passing messages characteristic of recursion in the computation of (factorial 3). Note that the computation proceeds from event to event along the "railroad tracks" in the diagram.



IV.4.e --- Characterization of Recursion as a Pattern of Passing Messages

Thus we see how recursion can be characterized as a pattern of passing messages using event diagrams. The characteristic feature is the build up of a chain of continuation actors each one of which knows only about the next and which eventually replies to the next with the answer. Notice that this characterization of recursion in terms of relations between events is independent of the syntax of the language for scripts which gives rise to the behavior. For example the same characterization would hold for a recursive implementation of factorial in ALGOL. The semantics of ALGOL can be defined using relations among events in a manner similar to the way in which the semantics of PLASMA is defined.

The existence of the actors labeled C' and C'' in the above diagram and the events in which they are the target are difficult to explain in terms of the above PLASMA script for factorial. In order to explain the origin of these actors and events, we need to explain more of the underlying implementation of PLASMA.

IV.5 --- Envelope Level Scripts

Thus far in our PLASMA scripts we have examined information communicated in the messages of envelopes. At this point we would like to introduce the envelope level which allows access to other information in the messengers of actor transmissions. Every messenger always contains (among other things) an actor which serves as the ENVELOPE. In turn every envelope always contains an actor which serves as the MESSAGE. Additionally REQUEST envelopes contain actors called CONTINUATIONS to which replies to the messages should be sent.

The reason that it is useful to introduce the envelope level transmitters and receivers into scripts is that otherwise much of the control structure (pattern of passing messages) has to remain implicit in something like an evaluator or a compiler. Envelope receivers and transmitters provide the mechanism for expressing more explicit scripts so that none of the processing or allocation of storage is going on behind the scenes.

Envelope receivers and transmitters are analogous to ordinary receivers and transmitters in many respects. They are intended to be used as a notation for writing scripts in which all the computational events and actors are explicitly shown. In this way the structure of simple control structures such as iteration and recursion can be explicitly characterized as patterns of passing messages.

PLASMA uses the syntactic convention of using the number of shafts on the transmitter and receive arrows to reflect the level at which the transmission is being referenced; one shaft meaning ordinary message level, and two shafts meaning envelope level. Thus:

<= is an (ordinary) message-level-transmitter, and
<== is a envelope-level-transmitter.

Similarly,

$\Rightarrow$ is an (ordinary) message-level-receiver, and
 $\Rightarrow\Rightarrow$ is an envelope-level-receiver.

Below we use this notation to make the message-passing underlying the implementation of PLASMA more explicit.

For example an ordinary message receiver which receives one argument n and replies with the value $(n + 1)$ written as

$(\Rightarrow [=n]$
 $(n + 1))$

can be written at the envelope level as follows:

$(\Rightarrow\Rightarrow (request: [=n] (reply-to: =c))$
 $(c \Leftarrow (reply: (n + 1))))$

IV.5.a --- A More Explicit Script for the Non-Iterative Factorial

The correspondence between the event diagram for the non-iterative implementation of factorial and its script can be made more apparent by using envelope transmitters and receivers to make the underlying implementation explicit. The script presented below is intended to explicate how the implementation of PLASMA actually works.

<pre> (factorial $\equiv$ ($\Rightarrow\Rightarrow$ (request: [=n] (reply-to: =c)) (rules n ($\Rightarrow$ 1 (c $\Leftarrow$ (reply: 1))) ($\Rightarrow$ (> 1) (factorial $\Leftarrow$ (request: [(n - 1)] (reply-to: ($\Rightarrow\Rightarrow$ (reply: =y) (c $\Leftarrow$ (reply: (y * n)))))))))) </pre>	<pre> ;factorial is defined to be ;receive a request to compute the value of factorial for ;an argument tuple whose only element is n and ;send the reply to the actor c ;the rules for n are ;if it is 1 then ;send c a reply envelope with message 1 ;else if it is greater than 1 ;send factorial a request ;with message (n - 1) and ;continuation the following actor ;if a reply envelope with message y is received ;then send c a reply envelope with message (y * n) </pre>
---	--

Notice that the above script specifies that before recursively calling factorial (in the case where $n \neq 1$), a new actor is created as the *reply-to* component of the envelope sent to factorial. This new actor is created with ACQUAINTANCES n and c and has the following SCRIPT:

```
(==> (reply: =y)
      (c <== (reply: (y * n))))
```

Operationally, the script says "for each reply *y* that is received, multiply it by *n* and send the resulting product as a reply to *c*".

IV.6 -- Iteration

It is well known that another, more efficient implementation of factorial uses iterative control structure. Event diagrams will be used as a tool to illustrate the behavior of this more efficient implementation of factorial. One idea for an iterative implementation is to gradually build up the product while counting down the argument --doing one multiply for each iteration. So we define an actor called loop which should be sent both the current accumulation (which is initially 1) and the current count (which is initially the input n) on each iteration. The obvious way to do this is to repeatedly send loop a sequence of the form [accumulation count].

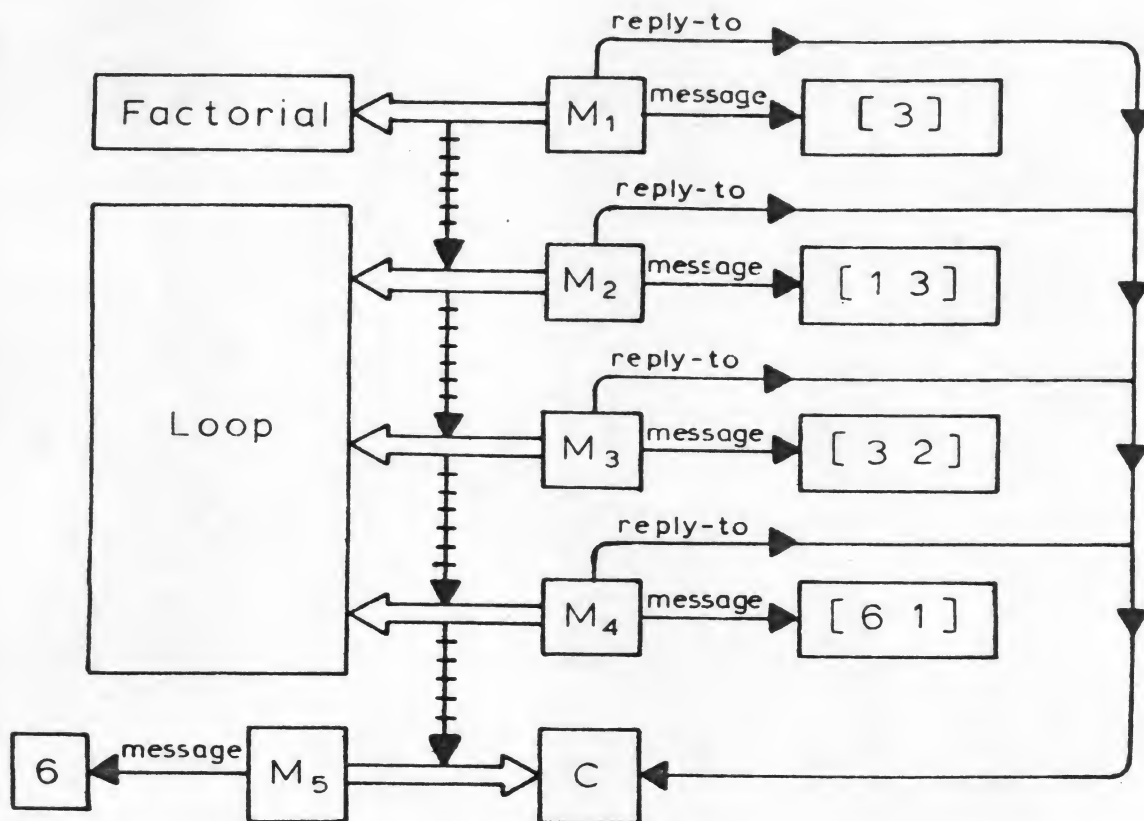
IV.6.a --- A Script for an Iterative Implementation of Factorial

<pre> (factorial ≡ (≡ [n] ([1 n] => (loop ≡ (≡ [accumulation =count] (rules count (≡ 1 accumulation) (≡ (> 1) (loop (accumulation * count) (count - 1)))))))))) </pre>	<pre> ;factorial is defined to be ;receive one argument and call it n ;send a 2-tuple with elements 1 and n to ;a newly created actor named loop which behaves as follows ;receive a 2-tuple as the current accumulated product and count ;the rules for the count are ; if it is 1 then ; return the accumulation ; else if it is greater than 1 ; send loop ;the accumulation times the count ;and the count minus one </pre>
--	---

Notice that the argument n is not an acquaintance of the actor loop in the iterative implementation of factorial. The rule for calculating the acquaintances from the script of an actor defined in PLASMA is very simple: the acquaintances of a newly created actor are the actors named by the free identifiers in the script at the time the actor is created. Instead of being an acquaintance, the actor n is sent to loop as the second element of the two tuple [1 n].

IV.6.b -- An Event Diagram for Iterative Factorial

The script given above will exhibit the behavior diagrammed below when factorial is sent the message [3]. This is an illustration of iteration as a pattern of passing messages. Note the repeated use of the actor C as a continuation in the envelopes used in the iterative implementation of factorial.



IV.6.c --- A More Explicit Script for Iterative Factorial

Notice that the above implementation of factorial definitely uses iterative (finite-state) control structure in the sense that it does not need any more memory than that needed for the values of count and accumulation. We now incorporate envelope transmitters and receivers to make the script of the iterative implementation of factorial more explicit. In this way the correspondence between the event diagram for the iterative implementation and its script becomes more apparent.

(factorial ≡	<i>;factorial is defined to be</i>
(≡≡> (request: [=n]	<i>;receive a request with argument tuple [n]</i>
(reply-to: =c))	<i>;and continuation c</i>
((request: [1 n] (reply-to: c)) =>	<i>;send a request with argument tuple [1 n] and</i>
	<i>;continuation c to the following newly created actor</i>
(loop ≡	<i>;named loop</i>
(≡≡> (request: [=accumulation =count] (reply-to: =d))	<i>;such that if a request is received with</i>
	<i>;message containing the accumulation and count</i>
	<i>;and continuation d</i>
(rules count	<i>;checks the count</i>
(≡> 1	<i>;to see if it is 1</i>
(d <= (reply: accumulation)))	<i>;if so it sends the accumulation as a reply to d</i>
(≡> (> 1)	<i>;else if it is greater than 1 then</i>
(loop <=	<i>;send loop a request with</i>
(request: [(accumulation * count) (count - 1)]	<i>;the appropriate message</i>
(reply-to: d)))))))))	<i>;and the continuation d</i>

The reason that this is iterative is that loop always passes along the same continuation actor that it receives with the message. The only continuation it needs, and therefore the only one that it holds onto, is the one contained in the original envelope that was sent to factorial. The loop sends its answer to that continuation directly when it is done. Thus no extra storage is needed going around the loop. Furthermore, in this implementation of iteration there are no side effects which change the behavior of any actor. If the user wants, she can keep a complete history of all the events in her computation and be confident that no information has been lost. Actor semantics account for the iterative behavior of the above implementation of factorial without having to appeal to external implicit mechanism such as an interpreter or any kind of external storage mechanism such as activation records. All the behavior of the system is accounted for by the behavior of actors when they are sent messages. Furthermore all of the storage is accounted for by the actors shown in the event diagrams. Event diagrams show how PLASMA is actually implemented using actors. The actor model provides a complete self-contained rigorous theory of iteration as a pattern of passing messages. It provides an explanation for the semantics behind the optimization rule used by many compilers that all "tail recursive" self-referential definitions can be compiled using special iteration primitives such as "while" loops, "do" loops, etc.

IV.6.d -- Meaning of "Recursion"

The term **RECURSIVE** has come to have at least three different meanings in computer science:

- 1: Effectively computable as in "recursive function theory"
- 2: Self-referential as in "factorial can be defined recursively in terms of itself"
- 3: Non-iterative as in "recursive functions use up more push-down stack when they call each other whereas iterative loops do not".

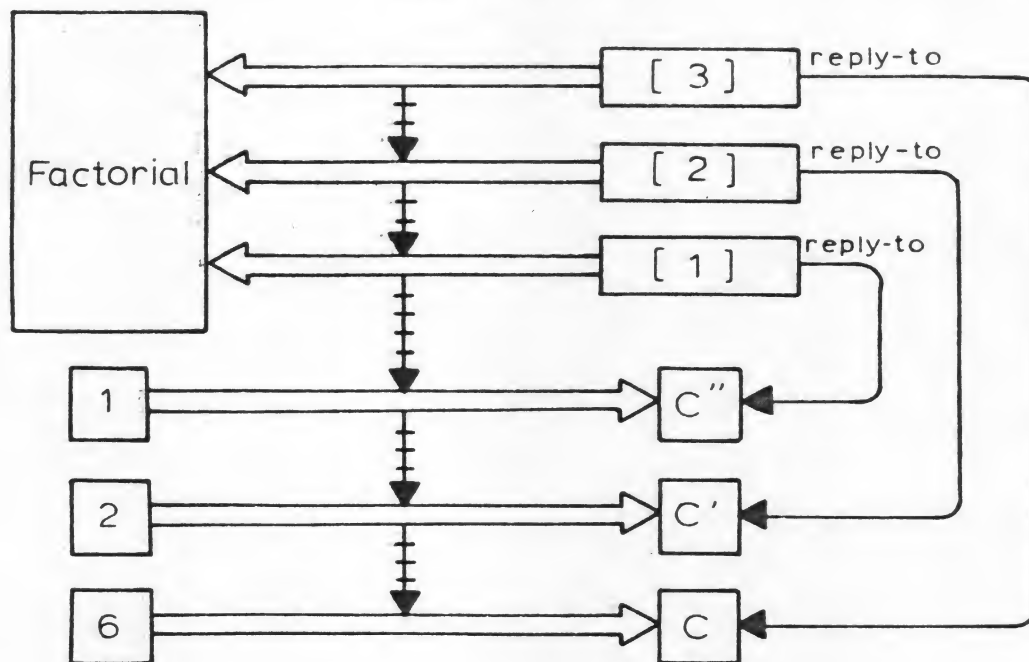
Both the iterative and non-iterative definitions for factorial which we have presented are self-referential. However, only the non-iterative implementation is "recursive" in the third sense of the word.

Using factorial as a simple example, we have shown how the actor message passing model can be used to give additional precision to fundamental concepts in computer science.

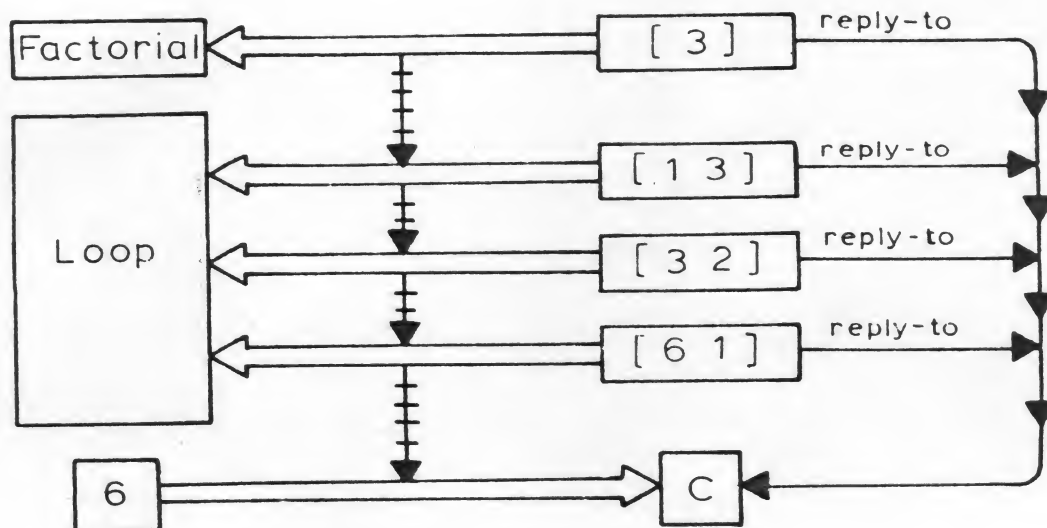
IV.7 -- Comparison of Recursion and Iteration

Below we present abstracted versions of the event diagrams for the iterative and non-iterative implementations of factorial when called with 3 as an argument. In the diagrams below the message is shown inside the messenger in order to more strongly bring out the pattern of message passing.

RECURSION



ITERATION



SECTION V --- EFFICIENCY and INTELLIGIBILITY

V.1 -- Modular Distribution of Knowledge

Since the defining characteristic of actors is that they send and receive messages, they are relatively unbiased with respect to assumptions about control structure and the distinction between data and operators. The neutrality on the issue of division of knowledge between data structure and operators can be seen in the various ways in which one can distribute information in an actor system. How one might choose to distribute it depends on one's purposes and the various uses to which the knowledge can be put. Often it is desirable to represent knowledge redundantly with different uses of the same knowledge appearing in several guises in several different places. The point is that the actors allow distribution of knowledge in any way that is useful.

Early Artificial Intelligence programs were mainly organized as multi-pass heuristic programs consisting of a pass of information gathering, a pass of constraint analysis, and a pass of hypothesis formation. It is now generally recognized that multi-pass organizations of this kind are inflexible because it is often necessary for information to flow across these boundaries in both directions in a dialogue at all stages of the processing.

V.2 -- Non-hairy Control Structure

One of the most important results that has emerged from the development of actor semantics has been the further development of techniques to semantically analyze or synthesize control structures as a patterns of passing messages. As a result of this work, we have found that we can do without the paraphernalia of "hairy control structure" (such as possibility lists, non-local gotos, and assignments of values to the internal variables of other procedures in CONNIVER). None of the accouterments of "hairy control structure" seem to be necessary for communication among the plans of a high-level goal-oriented formalism. In particular "hairy control structure" is not needed to deal effectively and efficiently with anomalies and complaints encountered in the course of attempting to mechanize problem solving in such a formalism. The conventions of ordinary message-passing seem to provide a better structured, more intuitive, foundation for constructing the communication systems needed for expert problem-solving modules to cooperate effectively.

We have discovered a syntactic transformation by which it is possible to convert a program which uses hairy control structure into an equivalent program that uses ordinary message passing. The first step of the transformation is to convert each ordinary message receiver $\Rightarrow$ into the form $\Rightarrow\Rightarrow$ and each ordinary message transmitter $\Rightarrow$ into the form $\Rightarrow\Rightarrow$ using the techniques used in the examples above. The next step then simply to convert each envelope level receiver $\Rightarrow\Rightarrow$ into $\Rightarrow$ and each each envelope level transmitter $\Rightarrow\Rightarrow$ into $\Rightarrow$. The result is a program which make no use of hairy control structure.

However, it is not recommended that the above method be used to convert programs that use hairy control structure. The best way to achieve an efficient modular implementation of a problem solver is to reason directly in terms of the behavior required to solve the problem. It is highly undesirable to take a program that is difficult to understand because of the use of hairy control structure and "improve" it by eliminating the hairy control structure by a local syntactic transformation such as the one discussed above. In general such local transformations make badly structured programs worse instead of better.

We will present two examples of problems where hairy control structure was originally used to implement a difficult problem. As the problem to be solved has become better understood, more intelligible solutions which do not involve hairy control structure have been developed.

V.3 --- Gaining Efficiency thru Progressive Refinement

Efficient implementations of systems are usually most easily arrived at by beginning with a high-level goal-oriented plan and then progressively refining using specific domain-dependent knowledge. For example a simple recursive implementation for computing $\text{base}^{\text{exponent}}$ is given below:

```
(integer-exponentiation ≡
  (⇒ [=base =exponent]
    (rules exponent
      (⇒ 0
        1)
      (else
        (base * (integer-exponentiation base (exponent - 1)))))))
```

In the above example we have made use of an expression of the form

(else body)

as a convenient mnemonic abbreviation for

(⇒ ? body)

making use of the fact that the pattern ? will match anything.

The above plan is too inefficient to use to calculate large exponents. However, we do not intend to use it for this purpose! Instead of executing the plan, we propose to refine it to make it more efficient. These refinements have been accomplished by using a great deal of mathematical and problem solving knowledge.

The efficiency of the exponentiation routine can be improved by transforming it into an iterative form using the fact that integer multiplication is associative:


```

(integer-exponentiation ≡
  (≡) [=base =exponent]
    ([exponent 1] =>
      (till-exponent-zero ≡
        (≡) [=e =accumulation]
          (rules e
            (≡) 0
              accumulation)
            (else
              (till-exponent-zero
                (e - 1)
                (accumulation * base))))))))

```

However, the above procedure is still not very efficient.

Notice that if exponent is an even integer then

$$\text{base}^{\text{exponent}} = (\text{base} * \text{base})^{(\text{exponent} / 2)}$$

The above arithmetical fact can be used as the basis for making a faster exponentiation routine:

```

(fast-exponentiation ≡
  (≡) [=base =exponent]
    ([base exponent 1] =>
      (till-exponent-zero ≡
        (≡) [=b =e =accumulation]
          (rules e
            (≡) 0
              accumulation)
            (≡) (even)
              (till-exponent-zero
                (b * b)
                (e / 2)
                accumulation))
            (else
              (till-exponent-zero
                b
                (e - 1)
                (b * accumulation))))))

```

This last refinement is probably fast enough for most practical purposes. However, John Reynolds has pointed out that the above program is still inefficient in two ways:

After it is determined that the exponent is odd, when the loop is continued it is unnecessary to test that $(\text{exponent} - 1)$ is even.

After it is determined that the exponent is non-zero but even, when the loop is continued it is unnecessary to test that $(\text{exponent} / 2)$ is non-zero.

Reynolds showed how these inefficiencies could be removed by the use of assignment statements and *gotos*.

The double testing is easily eliminated in PLASMA by simply defining two auxiliary actors which handle positive and even exponents as special cases. This example demonstrates how the underlying strategies of optimizations can be captured by reasoning in terms of message-passing.

```
(faster-exponentiation ≡
  (≡> [=base =exponent]
    (let
      (positive-exponent ≡
        (≡> [=b =e =accumulation]
          (rules e
            (≡> (even)
              (positive-exponent
                (b * b)
                (e / 2)
                accumulation)))
            (else
              (even-exponent
                b
                (e - 1)
                (b * accumulation))))))
      (even-exponent ≡
        (≡> [=b =e =accumulation]
          (rules e
            (≡> 0
              accumulation)
            (else
              (positive-exponent
                (b * b)
                (e / 2)
                accumulation))))))
    then
      (rules exponent
        (≡> 0
          1)
        (else
          (positive-exponent base exponent 1))))))
```

The point of this example is that viewing control structure as a pattern of passing messages can be used to motivate optimizations that improve efficiency. A good programming methodology involves writing high-level goal-oriented plans to specify a task followed by progressively refining these plans to obtain efficient implementations. To support a programming methodology based on progressive refinement, it is necessary to have a unified coherent formalism which can encompass the necessary range of plans. The formalism needs to be sufficiently powerful to represent any potential optimization so that the complexity and efficiency of the optimization can be calculated.

V.4 -- Generators

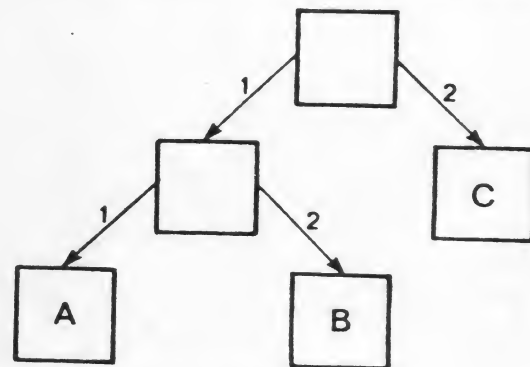
In knowledge based systems, it is unreasonable to store all the implications of the knowledge available at a given time. Explicitly storing the answers to all possible questions instead of incrementally generating them as they are needed is not only extremely inefficient since most of them may never be needed, but may in fact be impossible. For example expanding out all the possible games of chess before making the first move is clearly infeasible. Therefore it must be possible to incrementally generate implications as needed in order to answer questions.

In order to deal with this problem Newell, Shaw, Simon introduced a form of generators into their Information Processing Language. Since that time, the concept has undergone considerable further development. In terms of actors the idea is to construct a sequence s which behaves like a sequence of the possible answers to some question. The trick is that s does not physically contain all the answers but rather generates them incrementally as needed. To make this discussion more concrete we present a simple problem that illustrates how generators can be conveniently implemented in PLASMA.

We will assume that we have some actors called trees such that each tree is either of the form (terminal: $\underline{T}$) where $\underline{T}$ is the terminal symbol, or of the form (non-terminal: $\underline{L} \underline{R}$) where $\underline{L}$ and $\underline{R}$ are left and right sub-trees.

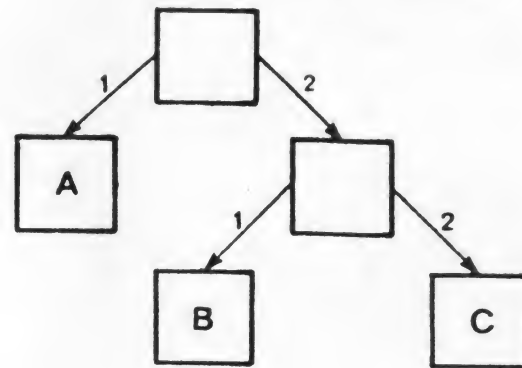
For example the tree

(non-terminal:
(non-terminal: (terminal: A) (terminal: B))
(terminal: C))



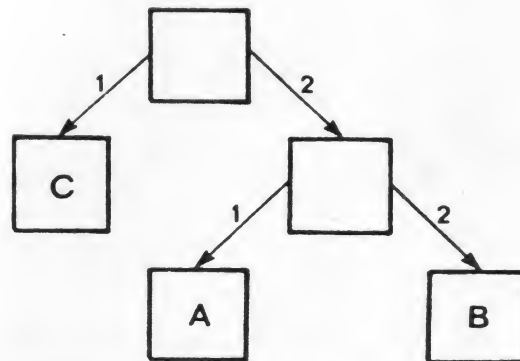
has the following fringe (sequence of terminals in left to right order) [A B C]

as does the following tree:



(non-terminal:
 (terminal: A)
 (non-terminal: (terminal: B) (terminal: C)))

whereas the following tree



(non-terminal:
 (terminal: C)
 (non-terminal: (terminal: A) (terminal: B)))

has [C A B] as its fringe.

The problem is to define the actor *fringe* so that for any tree *T*, (*fringe T*) behaves like a sequence of the terminal elements of *T*. There are two important properties that characterize the behavior of *fringe*. First, *fringe* of a terminal node must behave like a sequence with one element

$$(\text{fringe } (\text{terminal: } T)) \sim [T]$$

The symbol $\sim$ is used to denote behavioral equivalence of actors. Second, *fringe* of a non-terminal node must behave like the sequence produced by concatenating the fringe of the left sub-node and the fringe of the right subnode:

$$(\text{fringe } (\text{non-terminal: } L \ R)) \sim [!(\text{fringe } L) \ !(\text{fringe } R)]$$

The above specification makes use of the unpack operator **!** of PLASMA which is explained in the appendix.

V.4.a — A High-Level Implementation

From the above behavioral specifications we can immediately derive the following implementation of *fringe*:

```

(fringe ≡
  (⇒ [=the-tree]
    (rules the-tree
      (⇒ (terminal: =T)
        [T])
      (⇒ (non-terminal: =L =R)
        [!(fringe L) !(fringe R)]))))

```

;the behavior of fringe is defined to be
;whenever it receives a tree
;the rules for the tree are
;if is a terminal T
;then the fringe is a sequence whose only element is T
;else the tree must be a non-terminal
;and the fringe of the tree is
;the fringe of its left sub-tree concatenated with
;the fringe of its right sub-tree

Unfortunately, the above implementation is not incremental because it immediately looks at all the nodes of the tree and thus is exponentially inefficient. The above definition of *fringe* is still very much a specification of what *fringe* is supposed to do as opposed to a detailed specification of how to efficiently accomplish the task. This lack of concern with the details of implementation is the chief advantage (and at the same time the chief disadvantage) of high-level implementations.

V.4.b — An Incremental Implementation

Incremental generation amounts to adopting a "wait and see" approach as to whether the rest of the elements will be needed. The above implementation of *fringe* can be refined to be incremental by use of the *delay* operator. Readers who are not familiar with the *delay* operator of PLASMA should consult the appendix.

```

(fringe ≡
  (⇒ [=the-tree]
    (rules the-tree
      (⇒ (terminal: =T)
        [T])
      (⇒ (non-terminal: =L =R)
        [!(delay (fringe L)) !(delay (fringe R))]))))

```

;the behavior of fringe is defined to be
;whenever it receives a tree
;the rules for the tree are
;if is a terminal T
;then the fringe is a sequence whose only element is T
;else the tree must be a non-terminal
;and the fringe of the tree is
;the fringe of its left sub-tree concatenated with
;the fringe of its right sub-tree

The "wait and see" approach is not always the most efficient implementation for every problem. In particular often there is a space-time trade-off in the use of the *delay* operator. In many cases it is more efficient to simply compute an expression *E* immediately than to wait by the use of (*delay E*) since the latter can cause the retention of extra unnecessary storage. For example consider the following definition:

```

(if =
  (=> [=x =h]
    (rules x
      (=> (< 3)
        0)
      (else
        h))))

```

Notice that the expression (if 2 HUGE) immediately evaluates to 0 whereas the expression (delay (if 2 HUGE)) is an arbitrarily large amount of storage which will eventually evaluate to 0. The reader might consider how the efficiency of the implementation of the *delay* operator can be improved using partial evaluation.

An additional complexity is that PLASMA uses incremental sequences to implement pattern directed retrieval from a data base. This data base must have side-effects because it is used to implement communicating parallel processes [Greif and Hewitt 1975]. In this application the "do it now" and "wait and see" implementations can result in different sequences of values! In order to make interprocess communication work properly, careful control must be maintained over when delays are introduced into PLASMA scripts. This issue arises in the implementation of shared resources whose integrity must be protected as they are used by communicating parallel processes. For this reason PLASMA has been not been designed to use the delay rule for evaluation as the default evaluation mechanism as has been proposed for lambda calculus languages by Church, Cadiou, Vuillemin, Wadsworth, Henderson and Morris, and Friedman and Wise. Carried to its logical extreme the ultimate form of the uniform delay rule is to never compute the value of any expression unless the value is needed for output to the external environment!

SECTION VI --- The LAMBDA CALCULUS of CHURCH

As we have explicitly acknowledged in our previous papers, the development of PLASMA and the actor model of computation has been strongly influenced by the lambda calculus and by the work of numerous researchers who have studied it. The lambda calculus of Church is a suitable formalism for studying the behavior of effectively computable functions.

In our research we have attempted to constructively build on this previous work by developing a problem solving formalism and semantic model for actors such as cells, serializers, and funnels which do not behave like mathematical functions. In the sections below we investigate the different ways that previous researchers have used the lambda calculus as a formalism for studying the semantics of procedures.

The actor model of computation is based on incidental and causal relations among events where each event is defined by the act of sending one actor to another. Thus it is incorrect to speak of an "actors interpreter" because a semantic model does not specify a language which can be executed. The relationship between actors and PLASMA is analogous to the relationship between mathematical functions and the lambda calculus. Although there is a well developed mathematical theory of functions as sets of ordered pairs, there is no such thing as a "functions interpreter". The lambda calculus is just one of many possible languages which can be used to define the behavior of mathematical functions. Similarly, PLASMA is just one of many possible languages that can be used to define the behavior of actors.

In some useless sense all programming languages are equivalent. It is possible to simulate the behavior of any programming language using any other programming language in common use. Naively it might be thought that ALGOL is "more powerful" than FORTRAN because ALGOL has recursion and FORTRAN doesn't. However, there is a programming style in FORTRAN which enables recursive programs to be written in FORTRAN corresponding very closely to the way in which the programs would be written in ALGOL. The simulation involves allocating a large array to hold the temporary values needed in recursion. Similarly it is possible to simulate the behavior of PLASMA using a lambda calculus interpreter. The table below gives a simulation method for important behaviors of actors:

BEHAVIOR	PLASMA PRIMITIVE	LAMBDA CALCULUS SIMULATION TECHNIQUE
mutual-reference	labels	Y operator
side-effects	cell	"global state" of memory
synchronization	serializer	"global oracle"
parallelism	funnel	"global state" of program counters

All of the above simulation techniques work by systematically adding extra arguments to lambda

expressions. To simulate cells [Scott-Strachey] an extra argument is added to every lambda expression which is to be bound to a lambda expression which contains the "current contents" of all the cells on all the machines of the system. An assignment of new contents to a cell is simulated by constructing a new lambda expression which simulates the "next global state" of all the cells on the machines. Similarly to simulate synchronization an extra argument is added to every lambda expression which is to be bound to a lambda expression which simulates the "next" instruction to be executed on one of the machines executing in parallel. Thus the lambda calculus can be used to simulate the behavior of an actor system running on a network of machines executing in parallel. The lambda calculus simulation approach attempts to model all behavior by reduction to lambda abstraction and application. This raises an important question:

For what purposes is lambda calculus simulation a useful model of computation?

The answer to this question is currently under investigation by many researchers. We suspect that it will be several more years before researchers have reached a consensus of opinion on the question. However, we can make a few preliminary remarks that bear on what the ultimate answer might be.

Simulation using lambda expressions does not correspond very closely to the mechanisms that are actually used to implement communicating parallel processes on a network of machines executing in parallel. Networks of machines will soon become very common because of the rapidly decreasing cost of processors and rapid development of technologies to inexpensively provide high-bandwidth connections between machines.

PLASMA attempts to provide modular primitives which are intended to be used to implement abstractions that manifest useful problem solving behaviors such as communicating parallel processes. Within the actor model of computation, the behaviors of primitives such as cells, serializers, and funnels are axiomatized using incidental and causal relations among events. The actor model is intended to serve as the semantic foundation for a Programming Apprentice that supports an evolutionary behavioral programming methodology. In order for a Programming Apprentice to communicate effectively with the programmers building a system, it needs a semantic model which closely corresponds to the way in which programmers think about their computations. The actor message-passing model corresponds closely to the mechanisms that are actually used to implement communicating parallel processes on networks of machines.

SECTION VII --- FUTURE WORK**VII.1 -- Applications**

The PLASMA system described in this paper is currently being implemented at the MIT Artificial Intelligence Laboratory. In the spring of 1975, PLASMA was defined meta-circularly in terms of itself and then translated by hand into LISP ~~using~~ making use of LISP macros written by Russ Atkinson that make LISP resemble a subset of PLASMA. In the fall semester of 1975 the translation was completed and brought into an efficient running state by Howie Shrobe. However, more work is needed before it will be usable for writing large systems. This implementation [which has modularity and good human engineering as its chief design goals] is still under development. It is based on the actor transmission communication mechanism using primitive actors coded in LISP. The development of the actor metaphor will continue in the next year to gain some experience in using it for the following kinds of applications:

to implement a distributed symbolic evaluator for a Programming Apprentice [Hewitt and Smith 1975, Rich and Shrobe 1975, Yonezawa 1975]

to implement other procedural knowledge-based systems such as a stereotype-based visual perception system [McLennan 1975]

as a formalism for defining message passing systems to try out ideas for the modular distribution of knowledge for a society of communicating experts

to experiment with various scheduling and synchronization policies using serializers [Atkinson and Hewitt 1976]

as a basis for a flexible actor-based animation language [Kahn 1976]

VII.1a --- Incremental Perpetual Development

The development of any large system (viewed as a society) having a long useful life must be viewed as an incremental and evolutionary process. Development begins with specifications, plans, domain dependent knowledge, and scenarios for a large task. Attempts to use this information to create an implementation have the effect of causing revisions: additions, deletions, modifications, specializations, generalizations, etc. At all times in the perpetual development of the system the programmers are confronted with

1: A progression of more refined plans [programs, implementations, etc.] which partially accomplish some of the tasks specified.

2: Partial specifications [contracts, intentions, constraints, etc.] for some of the subtasks which are to be accomplished.

3: Partial justifications [proofs, demonstrations, analysis of dependencies] regarding how some of the plans satisfy some of their specifications.

4: Partial descriptions of some of the background knowledge [mathematical facts, physical laws, questions of interactive users, government regulations, etc.] of the environment in which the system will operate.

5: A collection of scenarios [at various articulations of detail] demonstrating how the system is supposed to work in concrete instances.

The success of an evolutionary behavioral modeling methodology is highly dependent on the development of competent Programming Apprentices [Hewitt and Smith 1975, Rich and Shrobe 1976, Yonezawa 1976] that help keep the above potentially disparate descriptions of a system coherently organized. The primary benefit of maintaining this coherence is not to prove once and for all that the implementation is CORRECT in any absolute sense. Changes in the environment external to the system will require that the system must either adapt its behavior to the changed circumstances or be supplanted. Rather the chief benefit of demonstrating the coherence of multiple descriptions of a system is to make the dependencies among the parts explicit so that the system can be readily adapted to the perpetually changing external environment. Already for many systems considerably more money is spent on modification and enhancement than on initial design and implementation.

VII.2 --- The Actor Problem-Solving Metaphor

The actor metaphor for problem solving is a large human scientific society: each actor is a scientist. Each has her own duties, specialties, and contracts. Control is decentralized among the actors. Communication is highly stylized and formal using messages that are sent to individual actors.

Problem solving proceeds by the attempts of experts to guess, or to conjecture, a plan for a solution followed by attempts to criticize the usually somewhat faulty initial plan. Plans for action are put forward for trial, to be eliminated or modified if not germane to the problem at hand. Tentative acceptance of a proposed plan must be combined with an ability to revise it if it is demonstrated to be infeasible. We make it our task to construct expert problem-solving modules to live in a world characterized by incomplete knowledge; to adjust themselves to it as well as they can; to take advantage of the opportunities they can find in it; and to solve the problem, if possible (they need not assume that it is), with the help of the knowledge available. If this is the task, then there is no more rational procedure than the method of planning, refining, and criticizing: of proposing new plans; progressively refining these plans to incorporate knowledge relevant to their execution, criticizing these refinements to expose their deficiencies; and of tentatively following them if they survive.

Newell (1962) points out two potential difficulties which must be dealt with by systems which adopt the actor problem solving methodology. First, the messages (carried by the messengers) must sometimes contain strategies, not just facts. They must be in the form of partial information that can be combined with other information available to the target actor. A good formal language must be developed for this kind of communication. The second potential difficulty is that a society operating in this fashion must not become a bureaucracy bogged down in sending messages back and forth without making any progress. We propose to rely on the critical nature of actors which are delegated subtasks to help control aimless thrashing.

We would like to emphasize that in the current state of the art only a small part of this metaphor can be realized in practice. At this point in time the metaphor serves mainly to provide suggestions of directions in which to work. Perhaps in the very far future it will be possible to construct computer systems which have a significant fraction of the expertise and communication ability of a small scientific subfield.

SECTION VIII --- ACKNOWLEDGEMENTS

The research reported in this paper was sponsored by the MIT Artificial Intelligence and Laboratory and Project MAC under the sponsorship of the Office of Naval Research.

Writing this paper would not have been possible without the generous help and encouragement of Marilyn McLennan. Many people have given us valuable feedback and criticism on the ideas in this paper. The detailed comments and criticisms of Robert Baron, Candy Bullwinkle, Henry Lieberman, Marilyn McLennan, Ron Pankiewicz, Chuck Rich, Bruce Schatz, and Brian Smith have vastly improved the presentation of the ideas in this paper. We would like to thank Hal Abelson, Russ Atkinson, Roger Banks, Edward Fredkin, Danny Hillis, Ben Kuipers, Chuck Reiger, Steve Saunders, Howie Shrobe, Brian Smith, Peter Szolovits, Jim Stansfield, Richard Steiger, Guy Steele, Richard Waters, and Aki Yonezawa for their comments and suggestions.

Our event diagrams and semantic definitions demonstrating that iteration and co-routine control structures can be efficiently implemented in PLASMA without using "hairy control structure" have been the subject of numerous lectures which we have given at M.I.T. and elsewhere in the last year. The event diagram for the recursive implementation of factorial appearing in this paper is a simplified version of the one presented by Richard Steiger in his masters thesis. They were presented in a tutorial lecture at the International Joint Conference on Artificial Intelligence held at Tbilisi in September 1975. The method described in this paper for doing iteration [published in the paper by Greif and Hewitt in the Conference Record of the January 1975 ACM Symposium on Principles of Programming Languages] has influenced Sussman and Steele to make the same method work for SCHEME.

The progress we have made on actors would have been completely impossible without the contributions and questions of numerous MIT students. Ben Kuipers, Howie Shrobe, Keith Nishihara, Brian Smith, Aki Yonezawa, Richard Steiger, and Peter Bishop, and Irene Greif have done much of the work in making actors intelligible and relevant to the problems of constructing knowledge-based systems.

Conversations with Alan Kay, John McCarthy, Alan Newell, and Seymour Papert were useful in getting us started on this line of research. Newell's thought provoking paper entitled "Some Problems of Basic Organization in Problem-Solving Programs" has inspired many of the ideas in this paper. Our research has concentrated on the development of a rigorous model of computation based on relationships among computational events. The development of this model has been greatly influenced by Seymour Papert's "little people" model of computation, a seminar given by Alan Kay at M.I.T. on an early version of SMALLTALK, and the work of Church, Fischer, Landin, McCarthy, Milner, Morris, Plotkin, Reynolds, Scott, Stoy, Strachey, Tennent, Wadsworth, etc. on formalisms based on the lambda calculus. The treatment of the behavior of sequences in this paper is an adaption of the "stream" concept of Landin and the generators of the IPL languages of Newell, Shaw, and Simon.

PLASMA has been designed to provide the basis for the implementation of a Programming

Control Structure

Apprentice for expert programmers. The behavioral programming methodology which PLASMA is intended to facilitate owes a tremendous intellectual debt to the concepts in SIMULA [Birtwistle et al. 1973, Palme 1973]. We are indebted to Alan Kay for calling our attention to these virtues of SIMULA.

The current implementation of PLASMA was designed by Carl Hewitt and has been implemented in LISP over the last year by a team of people whose principal members were Russ Atkinson, Tom Downey, Carl Hewitt, Marilyn McLennan, and Howie Shrobe. The implementation has been accomplished using a set of LISP macros implemented by Russ Atkinson that make LISP into a very limited subset of PLASMA. Howie Shrobe put the system together in the fall semester of 1975. This spring Marilyn McLennan has brought the system to a usable state. Tom Downey and Jerry Morrison have implemented a modular format printer for PLASMA programs. Carl Hewitt and Russ Atkinson have designed modular primitives for the implementation of parallelism and synchronization in PLASMA.

SECTION IX --- BIBLIOGRAPHY

- Atkinson, R. and Hewitt, C. "Synchronization in Actor Systems" Forthcoming. 1976.
- Barton, R. S. "Ideas for Computer Systems Organization: A Personal Survey" Software Engineering 1, Academic Press, 1970.
- Bishop, P. "Garbage Collection in a Very Large Address Space" MIT AI Working Paper III. September 1975.
- Birtwistle, Dahl, Myhrhaug, and Nygaard. SIMULA Begin Auerbach. 1973.
- Bobrow, D. and B. Wegbreit, "A Model for Control Structures for Artificial Intelligence Programming Languages" IJCAI-73, Stanford: Stanford University, August, 1973.
- Burge, W. H. "Stream Processing Functions" IBM Journal of Research and Development. Vol. 19. No. 1. January, 1975. pp. 12-25.
- Cadiou, J. M. "Recursive Definitions of Partial Functions and their Computations" Ph.D. Thesis, AIM-163, Stanford: Stanford University, April, 1972.
- Church, A. "The Calculi of Lambda Conversion" Annals of Mathematical Studies 6, Princeton University Press, 1941, 2nd edition 1951.
- Davies, D. J. M. "POPLER 1.5 Reference Manual" TPU Report No. 1. Theoretical Psychology Unit, School of Artificial Intelligence, University of Edinburgh. May, 1973.
- Dijkstra, E. W. "Notes on Structured Programming" August, 1969.
- Erman, L. D. and Lesser, V. "A Multi-level Organization for Problem Solving using Many, Diverse, Cooperating Sources of Knowledge. Proceedings of IJCAI-75. September, 1975.
- Evans, A. "PAL - A Language for Teaching Programming Linguistics", Proceedings of 23rd National Conference, 1968.
- Fischer, M. J. "Lambda Calculus Schemata" ACM Conference on Proving Assertions about Programs" SIGPLAN Notices. January 1972.
- Fisher, D. A. "Control Structures for Programming Languages" Ph. d.. Carnegie-Mellon University. 1970.
- Friedman and Wise "Cons Should Not Evaluate its Arguments" Indiana Technical Report 44. November, 1975.

Greif, I. "Semantics of Communicating Parallel Processes" Ph. D. M. I. T. September, 1975. Project MAC Technical Report TR-154.

Greif, I. and C. Hewitt "Actor Semantics of PLANNER-73" Proceedings of ACM SIGPLAN-SIGACT Conference, Palo Alto, January, 1975.

Henderson, P. and Morris, J. H. "A Lazy Evaluator" SIGPLAN-SIGACT Symposium on Principles of Programming Languages. Atlanta. January, 1976.

Herriot, R. G. "A uniform view of control structure in programming languages" Information Processing 74, Vol. 2, pp 175-444, August, 1974

Hewitt, C. "PLANNER: A Language for Manipulating Models and Proving Theorems in a Robot" IJCAI-69, Washington, D.C., May, 1969.

Hewitt, C. "Procedural Embedding of Knowledge in PLANNER" IJCAI-71, London, September, 1971.

Hewitt, C. "Protection and Synchronization in Actor Systems" Working Paper-83, Artificial Intelligence Laboratory, Cambridge: M.I.T., November, 1974. Revised December, 1975.

Hewitt, C., Bishop P., and R. Steiger, "A Universal Modular Actor Formalism for Artificial Intelligence" IJCAI-73, Stanford: Stanford University, August, 1973. pp. 235-245.

Hewitt, Carl and Smith, Brian. "Towards a Programming Apprentice" IEEE Transactions on Software Engineering. SE-1, 1, March 1975.

Hoare, C. A. R. "Monitors: An Operating System Structuring Concept" Stanford: Stanford University, 1973.

Kahn, K. M. "An Actor-Based Computer Animation Language" MIT A.I. Working Paper 120. February, 1976.

Kay, Alan C. "FLEX, A Flexible Extendible Language" Computer Science Dept. Technical Report 4-7. University of Utah. June, 1968.

Kay, Alan C. "Reactive Engine" Unpublished Ph. D. Thesis, Computer Science Department, University of Utah, 1970.

Knight, T. "CONS" M.I.T. A.I. Working paper 80. November, 1974.

Landin, P. J. "A Correspondence Between ALGOL 60 and Church's Lambda-Notation" CACM, February, 1965.

Learning Research Group "Personal Dynamic Media" Technical report. Xerox Palo Alto Research Center. 1976.

Lenat, D. B. "BEINGS: Knowledge as Interacting Experts" IJCAI-75. September, 1975.

McCarthy, J. P. W. Abrahams, D. J. Edwards, T. P. Hart, and M. I. Levin, "Lisp 1.5 Programmer's Manual", M. I. T. Press, Cambridge, August, 1962.

McDermott D. V., and G. J. Sussman "The Conniver Reference Manual" A.I. Memo No. 259, Cambridge: M.I.T., May, 1972.

McLennan, Marilyn. "Understanding Simple Plant Pictures" in Progress in Perception. D.A.I. Research Report No. 13. University of Edinburgh. December, 1975.

Milner, Robin. "Processes: A Mathematical Model of Computing Agents" Proceedings of Logic Colloquium. Bristol, 1974.

Newell, A. "Some Problems of Basic Organization in Problem-Solving Programs" Rand Corporation Memorandum RM-3283-PR. December, 1962.

Palme, J. "Protected Program Modules in SIMULA-67" Technical Report PB-224 776. Research Institute of National Defense. Stockholm. July 1973.

Reynolds, J. C. "GEDANKEN a Simple Typeless Language Based on the Principle of Completeness and the Reference Concept" CACM, 1970.

Reynolds, J. C. "Definitional Interpreters for Higher-Order Programming Languages" ACM National Convention, 1972.

Rich C. and Shrobe H. "Understanding LISP Programs: Towards a Programmer's Apprentice" Working Paper 82. December 1974.

Rulifson Johns F., J. A. Derksen and R. J. Waldinger "QA4: A Procedural Calculus for Intuitive Reasoning" Ph.d. Stanford: Stanford University, November 1972.

Steiger, R. "Actor Machine Architecture" M.S., Cambridge: M.I.T., June, 1974.

Sussman, G. J. and Steele, G. L. "SCHEME: An Interpreter for Extended Lambda Calculus" MIT AI Memo 349. December, 1975.

Yonezawa, A. "Symbolic Evaluation of Programs as an Aid to Program Construction" MIT AI Lab working paper. 1975.

Vuillemin, J. "Correct and Optimal Implementations of Recursion in a Simple Programming Language. Journal of Computer and System Sciences. vol 9. no 3. December 1974.

Wadsworth, Christopher. "Semantics and Pragmatics of the Lambda-calculus" Ph. D. Oxford. 1971.

Wirth, N. "Program Development by Stepwise Refinement" CACM 14, pp. 221-227. 1971.

SECTION X --- APPENDIX: Introduction to PLASMA**X.1 -- Sequences and Collections**

We will begin by presenting some very simple PLASMA scripts and gradually work our way up to more complicated examples.

Meta-syntactic variables will be underlined.

We note initially that [A₁ A₂ ... A_N] means an ordered sequence of the actors A₁ through A_N whereas {A₁ A₂ ... A_N} means a unordered collection of the actors A₁ through A_N. Thus [3 'b] is not equivalent to ['b 3] although {3 'b} is equivalent to {'b 3}. Also collections behave differently from mathematical sets in that {3 'b 3} is not equivalent to {3 'b} but is equivalent to {3 3 'b}.

Thus PLASMA has syntactic delimiters which are used consistently for the following different purposes:

- [...] delimits an ordered sequence of elements
- {...} delimits an unordered collection of elements
- (...) delimits an expression in PLASMA

X.2 -- Transmitters

A simple syntax for sending an actor M (called the message) to an actor T (called the target) is:

$$(T \Leftarrow M)$$

or the following, which is entirely equivalent³

$$(M \Rightarrow T)$$

Thus,

$$(['this 'is 'a 'simple 'sentence] \Rightarrow \text{parser})$$

will send a sequence of the five symbols 'this, 'is, 'a, 'simple, and 'sentence to the actor denoted by parser.

3: The reason for having two different syntactic forms for the transmission of a message is that often it is more readable to have the expression for the message before the expression for the target or vice versa. The difference is particularly noticeable when one is much smaller than the other.

Since it is very common to want to send a sequence of arguments to an actor, a simple syntactic form is needed for this purpose. For example the notation used above would require us to write $(+ \leq [x \ y \ z])$ in order to compute the sum of x , y , and z . whereas we would prefer use the syntax $(+ \ x \ y \ z)$.

In PLASMA, as in LISP, an expression of the form $(E_1 \ E_2 \ \dots \ E_n)$ ordinarily denotes an ordinary procedure call with procedure E_1 and arguments $E_2, \dots$, and E_n . Since PLASMA also uses parentheses as the delimiters of special syntactic forms, it needs to have some mechanism to distinguish special syntactic forms such as $(f \leq [3 \ 4])$ from ordinary procedure calls so that $\leq$ is not taken to be the second argument of f . PLASMA uses RESERVED SYMBOLS in parenthesized expressions for this purpose. For example both $\Rightarrow$ and $\leq$ are reserved symbols. Transmitters using the reserved symbols $\Rightarrow$ and $\leq$ are read as forms of the verb "SEND". For example $(f \leq [1 \ 3])$ would be read as " f is sent the sequence 1 3", or "a sequence of 1 and 3 is sent to f ".

For example

$(\text{factorial } 3)$	is equivalent to	$(\text{factorial } \leq [3])$
(generate)	is equivalent to	$([] \Rightarrow \text{generate})$

Note that when either of the transmitter arrows $\leq$ or $\Rightarrow$ is written out explicitly in a special syntactic form, there is always one expression before the arrow and one after it.

Also note that arithmetic can be expressed in infix notation as well as prefix notation. Arithmetic expressions are implemented in PLASMA by making arithmetic symbols such as $+$ and $*$ reserved symbols so that special modules associated with these symbols can process the expression in which they occur when the script is reduced.

The syntactic forms

$(\underline{\text{target}} \leq \underline{\text{message}})$ and $(\underline{\text{message}} \Rightarrow \underline{\text{target}})$

are designed to direct the eye of the reader along the normal flow of control of the message to the target. The transmitters of PLASMA are a generalization of the functional applications in the lambda calculus of Church which were defined in terms of substitution semantics. The semantics of transmitters are behaviorally defined in terms of events in the actor message-passing model.

X.3 -- Pattern Matching

Pattern matching is used in PLASMA to recognize actors which satisfy a simple description and to bind the answers to simple requests. The process is meant to be quite intuitive. For example The prefix = in front of an identifier name in a pattern can be used to bind the identifier to the corresponding object being matched. For example typing

```
(match [x=y] 10 [3 4])
```

can be used to bind x to 3 and y to 4

X.4 -- Receivers

Corresponding to the syntax for sending messages is a syntax for their reception. A PLASMA message-receiver has the following syntax:

(Ξ) pattern
body)

where the reserved symbol $\Rightarrow$ is read as "RECEIVE". Note the use of the three horizontal bars for the shaft a receive arrow as opposed to the use of two horizontal bars for a transmitter arrow. If an actor with the above definition is sent a message which matches pattern then body will be evaluated in the environment resulting from the pattern match. For example the PLASMA expression

$(([5\ 7] \Rightarrow$ *;send the tuple whose first element is 5 and second element 7*
 $(\Rightarrow [x = y]$ *;to a receiver which names the first element of the sequence received x and the second y*
 $(x + y)))$ *;and replies with the sum of x and y*

evaluates to 12.

For the sake of exposition we will call the actor that ($\Rightarrow$) pattern body creates a receiver. The behavior of the receiver is roughly as follows: when the receiver is sent a message, it matches it against the pattern. A **PATTERN** is an actor which decides whether it will match another actor called an object -- the process is asymmetric. If the match is unsuccessful, then the receiver complains that the message is not acceptable. If the match is successful, the pattern creates a new environment (which contains the bindings that resulted from the matching process). The receiver then sends the body an eval message that contains the new environment.

The syntactic form for receivers

($\equiv$) **pattern body**)

Writing complicated patterns results in tortuous obscure code. Simple patterns are a good way to bind identifiers to values. Pattern matching in PLASMA is a generalization of the lambda calculus identifier binding mechanism. The semantics of receivers is behaviorally defined by axioms in terms of the actor message-passing model.

The evaluation of a receiver results in an actor which has as its script the receiver and as its acquaintances the actors bound to the free identifiers of the receiver. For example if we type

$$(a \equiv [(3 + 2) (3 - 2)])$$

then we will create an actor [5 1] which is called a in the current local environment in which we are working. If we then type

(f $\equiv$ $\begin{array}{l} (\Rightarrow [x] \\ (g\ x\ a))) \end{array}$;define f to be an actor which
;when it receives a sequence with one element which will be called x
;replies with g of x and a

an actor will be created which has [5 1] and the value of `g` as its acquaintances.

X.5 -- Conditionals

Conditionals in PLASMA take two standard forms which are closely related. One form conditionally tests the value of an expression, the other conditionally tests the incoming message. The first is known as the rules expression and has the form:

(rules an-expression

($\Rightarrow$ pattern₁
 body₁)

($\Rightarrow$ pattern₂
 body₂)

...

($\Rightarrow$ pattern_n
 body_n))

;the rules for the actor an-expression are

 ;if it matches pattern₁ then

 ;reply with the value of body₁

 ;else if it matches pattern₂ then

 ;reply with the value of body₂

 ;else it must match pattern_n so

 ;reply with the value of body_n

The expression is matched against the successive patterns until it matches one of them; then the corresponding body is evaluated in the environment resulting from the pattern match. For example,

(rules (3 + 4)

($\Rightarrow$ (even)

5)

($\Rightarrow$ =n

(2 * n)))

;the pattern (even) will match any even integer

;the pattern =n will match any actor and bind n to that actor

;return twice the value of n

evaluates to 14.

PLASMA uses a similar construct (called a **cases** statement) to conditionally dispatch on an incoming message.

(cases

($\Rightarrow$ pattern₁

body₁)

($\Rightarrow$ pattern₂

body₂)

...

($\Rightarrow$ pattern_n

body_n))

;the cases for a message sent to this actor are

;if the message matches pattern₁ then

;reply with the value of body₁

;else if the message matches pattern₂ then

;reply with the value of body₂

;else the message must match pattern_n so

;reply with the value of body_n

A message sent to an expression of the above form is matched directly against the successive patterns until a match is found, whereupon the corresponding body is evaluated in the environment which results from the match.

For example the following actor replies with **yes** to any even number it is sent; replies with **no** to any odd number; and is otherwise not-applicable.

(cases

($\Rightarrow$ (even)

yes)

($\Rightarrow$ (odd)

no))

X.6 -- Definitions

In general, typing an expression of the form

(name $\equiv$ definition)

will cause PLASMA to do its subsequent evaluations in an environment which has been extended by binding name to the value of definition.

For example the normal way to interactively define integer-exponentiation while working at a console would be to type:

```
(integer-exponentiation ≡
  (⇒ [=base =exponent]
    (rules exponent
      (⇒ 0
        1)
      (⇒ (> 0)
        (base * (integer-exponentiation base (exponent - 1))))))
    ;integer-exponentiation is defined to have the following behavior
    ;whenever it receives a sequence of two arguments called base and exponent
    ;the rules for the exponent are
    ;if it is 0 then
    ;reply that the answer is 1
    ;else if it is greater than 0 then
    ;the answer is the base times the base to the power of the exponent minus 1
```

As an obvious extension to our notation for definitions we allow a parenthesized expression on the left hand side of a definition. For example we can define integer exponentiation in terms of an infix operator as follows:

```
((=base to-integer-power =exponent) ≡
  (rules exponent
    (⇒ 0
      1)
    (⇒ (> 0)
      (base * (base to-integer-power (exponent - 1))))))
    ;an expression of the form (=base to-integer-power =exponent)
    ;is defined by the following behavior
    ;the rules for the exponent are
    ;if it is 0 then
    ;the answer is 1
    ;else if it is greater than 0 then
    ;the answer is the base times the base to the integer power of the exponent minus 1
```

Using the above definition (5 to-integer-power 3) evaluates to 125. In this way we can conveniently define new kinds of syntactic forms.

MUTUALLY REFERENTIAL DEFINITIONS are easy to make using the reserved symbol *let* as follows:

```
(let
  (name1 ≡ D1)
  (name2 ≡ D2)
  ...
  (namen ≡ Dn)
  then
  body)
```

which evaluates body in an environment with each name_i bound to the value of D_i. The equations are mutually referential in that any occurrence of a name_i within a D_k refers to D_i.

As a special case of the *let* construct we use

```
(name ≡ definition)
```

as an abbreviation for

```
(let
  (name = definition)
  then
  name)
```

Self-referential definitions are very useful in defining iterative, recursive, and co-routine control structures. They are also useful in defining data structures that need to know about themselves.

At this point, we have enumerated all the ways to bind identifiers in PLASMA. Note that the definition of every symbol is lexically scoped and that there are no "global variables".

X.7 -- Unpack

We will often make use of an extremely useful operator for sequences and collections called UNPACK which is abbreviated as an exclamation point: !expression is always equivalent to writing out all of the elements of the expression individually. Thus if s is bound to the sequence [3 4 5], then the value of [1 2 !s] is [1 2 3 4 5]. Thus if the sequence [10 20 30 40 50] is matched against the pattern [=x =y !=z], then x will be bound to 10, y will be bound to 20, and z will be bound to [30 40 50] in the environment which results from the match. Unpack is in essence the inverse of sequence brackets "[...]".

The unpack operator neatly cleans up the confusion in LISP between different ways to construct lists. Considering analogies between LISP lists and PLASMA sequences, the following similarities hold:

[x y z]	is analogous to	(list x y z)
[x !y]	is analogous to	(cons x y)
![x y]	is analogous to	(append x (list y))
![x !y]	is analogous to	(append x y)

The chief benefit of the unpack notation is that the programmer no longer needs to concentrate on how to construct the structure by deciding whether to use CONS, LIST, or APPEND. Instead she can concentrate on what the structure should be by writing a pattern of what it should look like. For example the following PLASMA expression

```
![a [b !c d] !e]
```

has the following LISP analog:

```

(append
  a
  (cons
    (cons
      b
      (append c (list d)))
    e))

```

X.8 -- Use of Sequences

Sequences are a useful mechanism for the implementation of the kind of dialogues needed in the implementation of knowledge-based systems. They provide a useful common interface for co-routine control structures. We shall bind the elements and sub-sequences using pattern matching. The following pattern will bind *f* to the first element of a sequence and *r* to the rest:

[*f* !=*r*]

For example if *s* is bound to the sequence [14 3 105] then typing the following expression in PLASMA

(match [*f* !=*r*] to *s*)

will bind *f* to 14 and bind *r* to [3 105].

As an example of the use of sequences, we define the function *sum-of* which calculates the sum of all the elements in a sequence:

<pre> (sum-of ≡ (⇒ [=the-sequence] (rules the-sequence (⇒ [] 0) (⇒ [=the-next !=the-rest] (the-next + (sum-of the-rest)))))) </pre>	<pre> ;define the function sum-of ;to receive a sequence ;the rules for the sequence are ;if the sequence is empty ;then the sum is 0 ;else bind the next and the rest ;then return the next plus the sum of the rest </pre>
---	--

For example (sum-of [1 4 9]) evaluates to 14.

It is easy to build sequences. For example the following definition defines finite sequences of consecutive decreasing squares.

```

(sequence-of-squares ≡
  (⇒ [=n]
    (rules n
      (⇒ 0
        [])
      (⇒ (> 0)
        [(n * n) !(sequence-of-squares (n - 1))])))
    ;the rules for n are
    ;if it is 0 then
    ;the answer is the empty sequence
    ;else if it is greater than 0 then
    ;the answer is a sequence with n2 followed the squares for n minus 1

```

For example typing the following expression into PLASMA

```
(match [=first !=rest] to (sequence-of-squares 4))
```

results in binding `first` to the value 16 and binding `rest` to the value [9 4 1]. Thus `(sum-of (sequence-of-squares 4))` evaluates to 30.

X.9 -- Delay

For many applications, it is more efficient to generate the squares in the sequence of squares incrementally adopting a "wait and see" approach as to whether the rest of the elements will be needed. To this end we introduce the *delay* operator which delays computation of the value of expression `E` until the value is needed. Suppose that `vo` is the value of the expression `(delay E)`. The value of `E` is not computed until the actor `vo` is sent a message. The first time `vo` is sent a message, the value of `E` is computed and remembered. Thereafter `vo` behaves exactly like the value of `E`. It is unreasonable to delay the evaluation of any expression which does not always evaluate to the same object.

The *delay* operator can be used to refine the implementation of `sequence-of-squares` to produce an incremental-version:

```

(incremental-sequence-of-squares ≡
  (⇒ [=n]
    (rules n
      (⇒ 0
        [])
      (⇒ (> 0)
        [n2 !(delay (incremental-sequence-of-squares (n - 1)))])))

```

Typing the following into PLASMA

```
(match [=f1 !=r1] to (incremental-sequence-of-squares 10))
```

will bind `f1` to 100 and bind `r1` to an actor which is behaviorally equivalent to

(delay (incremental-sequence-of-squares 9))

At this point in time the only square that has been computed is the square of 10. Typing

(match [=f2 !=r2] to r1)

will bind f2 to 81 and bind r2 to an actor which is behaviorally equivalent to

(delay (incremental-sequence-of-squares 8))

X.10 — Packagers

PACKAGERS are a primitive mechanism in PLASMA for packaging actors together. They are very useful for packaging up the parts of a message. For example the notation $[x_1 \dots x_n]$ for a sequence is really just syntactic sugar for the package *(sequence: $x_1 \dots x_n$)*. Thus evaluation of an ordinary function call of the form $(f \leftarrow [x_1 \dots x_n])$ sends a package which is the sequence of arguments to f . However, the use of positional notation within a sequence for the components of a message is neither mnemonic nor secure. The packagers of PLASMA allow the components of the package to be explicitly named and the physical representation to be hidden (for reasons of efficiency and cleanliness). They permit all of the components of the package which are of interest to be selected in parallel and the remainder of the components to be ignored (for reasons of modularity and extensibility). Additionally, packagers provide for both the privacy and security of messages since in order to have access to the contents of a package constructed by a particular packager, it is necessary to have access to that packager. Packagers are the primitive authentication mechanism of PLASMA. A packager can only be taken apart by the packager which constructed it.

To illustrate the use of packagers we shall define a packager for complex numbers. First we define packagers for the messages to which complex numbers must respond:

(packager (real-part?))

(packager (imaginary-part?))

To make these abbreviations more convenient to use we define the following abbreviations:

((real-part =z) $\equiv$

((real-part?) => z))

((imaginary-part =z) $\equiv$

((imaginary-part?) => z))

;the real-part of z is computed by

;sending a message asking z for its real part

;the imaginary-part of z is computed by

;send a message asking z for its imaginary part

Below we define a packager for complex numbers:


```

(packager (complex: (real: ?) (imaginary: ?))
          ;define a packager for complex numbers with real and imaginary components
(=> (complex: (real: =x) (imaginary: =y))
    (cases
      (=> (real-part?)
          x)
          ;if asked for the real part then
          ;return x
      (=> (imaginary-part?)
          y)
          ;if asked for the imaginary part then
          ;return y
      (=> (plus: =z)
          (complex:
            (real: (x + (real-part z)))
            (imaginary: (y + (imaginary-part z)))))
          ;if asked for the sum with z then
          ;return a complex number with
          ;real component the sum of x and the real part of z
          ;and imaginary component the sum of y and the imaginary part of z
      (=> (times: =z)
          (complex:
            (real: ((x * (real-part z)) - (y * (imaginary-part z))))
            (imaginary: ((x * (imaginary-part z)) + (y * (real-part z)))))))

```

Notice the use of the packager *complex*: both to construct complex numbers and to take them apart into their real and imaginary parts. The above implementation is inefficient because of all the message passing involved in computing the values of *(real-part z)* and *(imaginary-part z)* when doing addition and multiplication of complex numbers. For example in the above implementation two such messages are required to compute the sum in the following sub-expression of the above program:

```

(complex:
  (real: (x + (real-part z)))
  (imaginary: (y + (imaginary-part z))))

```

We will demonstrate how the efficiency can be improved in a purely mechanical way without diminishing the generality of the implementation. The first step is to collect statistics of executions to determine which actors are very frequently sending messages to other. This will soon reveal that the expression *(real-part z)* quite often results in sending the message *(real-part?)* to *z* where *z* is of the form

```
(complex: (real: rz) (imaginary: iz))
```

This suggests that special code for this case might be generated in-line to speed up the execution. Obviously the expression *(real-part z)* is completely equivalent to

```

(rules z
  (=> (complex: (real: =rz) (imaginary: =iz))
      (real-part (complex: (real: rz) (imaginary: iz)) ))
  (else
    (real-part z)))

```

By replacing *real-part* and *complex:* by their definitions and simplifying we obtain the following expression:

```
(rules z
  (=> (complex: (real: =rz) (imaginary: =iz))
    rz)
  (else
    (real-part z)))
```

By performing the above transformation on all expressions of the form *(real-part z)* and *(imaginary-part z)* and then pulling out common sub-expressions the following more efficient implementation of the package *complex:* has been derived:

```
(packager (complex: (real: ?) (imaginary: ?))
  ;define a more efficient packager for complex numbers with real and imaginary components
  (=> (complex: (real: =x) (imaginary: =y))
    (cases
      (=> (real-part?)
        x)
      (=> (imaginary-part?)
        y)
      (=> (plus: =z)
        (rules z
          (=> (complex: (real: =rz) (imaginary: =iz))
            (complex:
              (real: (x + rz))
              (imaginary: (y + iz))))
          (else
            (complex:
              (real: (x + (real-part z)))
              (imaginary: (y + (imaginary-part z)))))))
      (=> (times: =z)
        (rules z
          (=> (complex: (real: =rz) (imaginary: =iz))
            (complex:
              (real: ((x * rz) - (y * iz)))
              (imaginary: ((x * iz) + (y * rz))))
          (else
            (complex:
              (real: ((x * (real-part z)) - (y * (imaginary-part z))))
              (imaginary: ((x * (imaginary-part z)) + (y * (real-part z))))))))))
```

Note that PLASMA is ideally suited for the above kind of optimization by in-line substitution because identifiers in PLASMA (unlike many other languages) are completely transparent. An occurrence of identifier in PLASMA serves only to name the actor to which it is bound. In-line substitution is not always valid in languages like LISP 1.5 because of the SET primitive in the language.

The presence of a primitive like SET (and other similar primitives in other LISP-like languages) makes optimization much more difficult.